

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/lm\_diags.c

DATE 5/23/89

PAGE #

TIME 4:41:14 pm

9/20

```

LINE # SOURCE TEXT
961 /* returns the interrupt character if in the buffer. */
962 /* or returns the first character in the buffer */
963 /* or returns 0 if no key */
964
965
966 _check_key()
967 {
968     register int retval;
969
970 #ifdef CPU_DIAGS
971     if (fifo_task == RECEIVE_TASK_ID) {
972         retval = _net_check_key();
973     } else
974     {
975         retval = _serial_check_key();
976     }
977 #else CPU_DIAGS
978     retval = _serial_check_key();
979 #endif CPU_DIAGS
980     if (retval == INTR_CHARACTER) intr();
981     return retval;
982 }
983
984 _serial_check_key()
985 {
986 #ifdef CPU_DIAGS
987     struct fifo_entry fifo;
988     register int retval;
989     register int i;
990
991     fifo.fifo_no = RX_FIFO;
992     retval = fifo_inquiry(&fifo);
993
994     if (retval) {
995         retval = _serial_get_key();
996         while (fifo_inquiry(&fifo)) {
997             i = _serial_get_key();
998             if (i == INTR_CHARACTER)
999                 retval = i;
1000         }
1001         if (i == ctrl(s)) {
1002             (void) _serial_get_key();
1003             /* simple "S" strategy: any char resumes printing */
1004         }
1005     }
1006     return retval;
1007 #else CPU_DIAGS
1008     register int retval = _check_key();
1009     register int i;
1010
1011     if (retval) {
1012         retval = _get_key();
1013         while (_check_key()) {
1014             i = _get_key();
1015             if (i == INTR_CHARACTER)
1016                 retval = i;
1017         }
1018         if (i == ctrl(s)) (void) _get_key(); /* simple "S" strategy:
1019                                             any char resumes printing */
1020     }
1021     return retval;
1022 #endif CPU_DIAGS
1023 }
1024
1025 /* parameter parsing routines */
1026
1027 #define CMDTOKEN_NOT_BUILT_IN 0
1028 #define CMDTOKEN_quit 1
1029 #define CMDTOKEN_all 2
1030 #define CMDTOKEN_exit 3
1031 #define CMDTOKEN_help 4
1032 #define CMDTOKEN_help 5
1033 #define CMDTOKEN_wizard 6
1034 #define CMDTOKEN_burnin 7
1035
1036 #define TOKEN_UNDEFINED 0
1037 #define TOKEN_max_cycles 1
1038 #define TOKEN_max_errors 2
1039 #define TOKEN_max_warnings 3
1040 #define TOKEN_max_messages 4
1041 #define TOKEN_summary_count 5
1042 #define TOKEN_errors_on 6
1043 #define TOKEN_warnings_on 7
1044 #define TOKEN_messages_on 8
1045 #define TOKEN_banner_on 9
1046 #define TOKEN_print_depth 10
1047 #define TOKEN_lm_fast_test 11
1048 #define TOKEN_forever 12
1049 #define TOKEN_output 13
1050 #define TOKEN_COUNT 14
1051 #define TOKEN_max_failures 15
1052
1053 #define DFLT_max_cycles 1
1054 #define DFLT_max_errors 1
1055 #define DFLT_max_failures 0
1056 #define DFLT_max_warnings 0
1057 #define DFLT_max_messages 0
1058 #define DFLT_summary_count 0
1059 #define DFLT_errors_on LM_FALSE
1060 #define DFLT_warnings_on LM_FALSE
1061 #define DFLT_messages_on LM_FALSE
1062 #define DFLT_banner_on LM_TRUE
1063 #define DFLT_print_depth 20
1064 #define DFLT_lm_fast_test LM_TRUE
1065 #define DFLT_forever LM_FALSE
1066
1067 char *get_identifier();
1068
1069 struct token {
1070     char *name;
1071     long length;
1072     long token;
1073 };
1074
1075 struct token cmdtokens[] = {
1076     { "all", 1, CMDTOKEN_all },
1077     { "burnin", 4, CMDTOKEN_burnin },
1078     { "quit", 1, CMDTOKEN_quit },
1079     { "exit", 1, CMDTOKEN_exit },
1080     { "help", 1, CMDTOKEN_help },

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/lm_diags.c		DATE 5/23/89	PAGE # 10/21
LINE #		SOURCE TEXT			
1081	{ "main",	1,	CMTOKEN_main},		
1082	{ "wizard",	3,	CMTOKEN_wizard},		
1083	0				
1084	};				
1085					
1086	struct token new_options[] = {				
1087	{ "continuous",	1,	TOKEN_forever},		
1088	{ "error_count",	1,	TOKEN_max_errors},		
1089	{ "failure_count",	1,	TOKEN_max_failures},		
1090	{ "fast_test",	4,	TOKEN_lm_fast_test},		
1091	{ "message_count",	1,	TOKEN_max_messages},		
1092	{ "print_banner",	7,	TOKEN_banner_on},		
1093	{ "print_depth",	7,	TOKEN_print_depth},		
1094	{ "print_errors",	7,	TOKEN_errors_on},		
1095	{ "print_messages",	7,	TOKEN_messages_on},		
1096	{ "print_warnings",	7,	TOKEN_warnings_on},		
1097	{ "p_banner",	2,	TOKEN_banner_on},		
1098	{ "p_depth",	2,	TOKEN_print_depth},		
1099	{ "p_error",	2,	TOKEN_errors_on},		
1100	{ "p_message",	2,	TOKEN_messages_on},		
1101	{ "p_warning",	2,	TOKEN_warnings_on},		
1102	{ "repeat",	1,	TOKEN_max_cycles},		
1103	{ "summary_print",	1,	TOKEN_summary_count},		
1104	{ "warning_count",	1,	TOKEN_max_warnings},		
1105	0				
1106	};				
1107					
1108	set_global_defaults()				
1109	{				
1110	max_cycles = DFLT_max_cycles;				
1111	max_errors = DFLT_max_errors;				
1112	max_failures = DFLT_max_failures;				
1113	max_warnings = DFLT_max_warnings;				
1114	max_messages = DFLT_max_messages;				
1115	summary_count = DFLT_summary_count;				
1116	errors_on = DFLT_errors_on;				
1117	warnings_on = DFLT_warnings_on;				
1118	messages_on = DFLT_messages_on;				
1119	banner_on = DFLT_banner_on;				
1120	print_depth = DFLT_print_depth;				
1121	lm_fast_test = DFLT_lm_fast_test;				
1122	forever = DFLT_forever;				
1123	}				
1124					
1125	/* execute_string() returns FAILURE only if we want to quit.*/				
1126					
1127	execute_string(string, menu)				
1128	register char *string;				
1129	LM_DIAG_MENU *menu;				
1130	#define is_specified(tokens) (specified & (1 << (tokens)))				
1131	{				
1132	INTR_INIT				
1133	register LM_DIAG_MENU_ITEM *menu_item;				
1134	long value, cmdtokens, tokens;				
1135	long reply;				
1136	char command[max_str];				
1137	long specified = 0; /* nothing specified yet.*/				
1138					
1139	set_global_defaults();				
1140	{				
1141	if (! (string = get_identifier(string, command)))				
1142	return SUCCESS;				
1143					
1144	cmdtokens = identify_token(command, &(cmdtokens[0]));				
1145					
1146	while (string = new_get_parameter(string, &tokens, &value)) {				
1147	switch (tokens) {				
1148	case TOKEN_max_cycles: max_cycles = value; break;				
1149	case TOKEN_max_errors: max_errors = value; break;				
1150	case TOKEN_max_failures: max_failures = value; break;				
1151	case TOKEN_max_warnings: max_warnings = value; break;				
1152	case TOKEN_max_messages: max_messages = value; break;				
1153	case TOKEN_summary_count: summary_count = value; break;				
1154	case TOKEN_errors_on: errors_on = value; break;				
1155	case TOKEN_warnings_on: warnings_on = value; break;				
1156	case TOKEN_messages_on: messages_on = value; break;				
1157	case TOKEN_banner_on: banner_on = value; break;				
1158	case TOKEN_print_depth: print_depth = value; break;				
1159	case TOKEN_lm_fast_test: lm_fast_test = value; break;				
1160	case TOKEN_forever: forever = value; break;				
1161	case TOKEN_UNDEFINED:				
1162	lm_banner("Syntax error: %s", string);				
1163	return SUCCESS;				
1164	default:				
1165	lm_banner("Token %d not implemented", tokens);				
1166	return SUCCESS;				
1167	}				
1168	specified  = (1 << tokens);				
1169	}				
1170					
1171	/*				
1172	Defaults:				
1173	{				
1174	1. Turn on all messages, warnings, errors only if a utility or				
1175	you are a wizard running a single test with no reps.				
1176	{				
1177	2. Turn off banners for multiply associated alltests.				
1178	{				
1179	3. pb-f implies pb-f implies pb-f implies pb-f				
1180	{				
1181	4. pb-t implies pb-t implies pb-t implies pb-t				
1182	{overrides 3 above}				
1183	{				
1184	{				
1185	/* See 1. above */				
1186	if (lm_wizard_mode && (max_cycles <= 1) &&				
1187	(forever == LM_FALSE) && (cmdtokens != CMTOKEN_all)) {				
1188	if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;				
1189	if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE;				
1190	if (!is_specified(TOKEN_messages_on)) messages_on = LM_TRUE;				
1191	}				
1192					
1193	/* 2. Default pb = f if r > 1 */				
1194	if ((max_cycles > 1)    (forever == LM_TRUE)) {				
1195	if (!is_specified(TOKEN_banner_on)) banner_on = LM_FALSE;				
1196	}				
1197					
1198	/* 3. pb-f implies pb-f implies pb-f implies pb-f */				
1199	if (!is_specified(TOKEN_banner_on) && (banner_on == LM_FALSE)) {				
1200	if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;				



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/lm\_diags.c

DATE 5/23/89  
TIME 4:41:14 pm

PAGE #  
11/22

```

1201         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1202         if (!is_specified(TOKEN_errors_on)) errors_on = LM_FALSE;
1203     }
1204     if (is_specified(TOKEN_errors_on) && (errors_on == LM_FALSE)) {
1205         if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1206         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1207     }
1208     if (is_specified(TOKEN_warnings_on) && (warnings_on == LM_FALSE)) {
1209         if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1210     }
1211 }
1212 /* 4. port implies port implies port implies port */
1213 if (is_specified(TOKEN_messages_on) && (messages_on == LM_TRUE)) {
1214     if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1215     if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1216     if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE;
1217 }
1218 if (is_specified(TOKEN_warnings_on) && (warnings_on == LM_TRUE)) {
1219     if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1220     if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1221 }
1222 if (is_specified(TOKEN_errors_on) && (errors_on == LM_TRUE)) {
1223     if (!is_specified(TOKEN_banner_on)) banner_on = LM_TRUE;
1224 }
1225 }
1226 switch (cmdtoken) {
1227 case CMDTOKEN_quit:
1228     return FAILURE; /* quit */
1229 case CMDTOKEN_all:
1230     lm_all_test(menu);
1231     break;
1232 case CMDTOKEN_banner:
1233     lm_banner("Banner is:\n");
1234     if (!is_specified(TOKEN_forever)) forever = LM_TRUE;
1235     if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1236     if (!is_specified(TOKEN_messages_on)) messages_on = LM_FALSE;
1237     if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_FALSE;
1238     if (!is_specified(TOKEN_banner_on)) banner_on = LM_FALSE;
1239     lm_all_test(menu);
1240     break;
1241 case CMDTOKEN_exit:
1242     exit_request();
1243     break;
1244 case CMDTOKEN_help:
1245     help();
1246     break;
1247 case CMDTOKEN_wizard:
1248     promote();
1249     break;
1250 case CMDTOKEN_main:
1251     lm_pop_up = LM_TRUE;
1252     return FAILURE; /* quit */
1253 default: /* not a built in command */
1254     menu_item = menu->menu_items;
1255     for (reply = 0;
1256         reply <= menu->number_of_items; ++reply, ++menu_item) {
1257         if (0 == strcmp(cmdtoken, menu_item->selection))
1258             break;
1259     }
1260     if ((reply == menu->number_of_items)
1261         || ((menu_item->attributes & LM_DIAG_wizard_mode)
1262             && !lm_wizard_mode)
1263         || ((menu_item->attributes & LM_DIAG_no_select)) {
1264         menu_line(LM_DIAG_ERROR, "Invalid selection: %s\n",
1265             cmdtoken);
1266         break;
1267     }
1268     if ((menu_item->attributes & LM_DIAG_dflt_msg_off)) {
1269         /* a utility */
1270         if (!is_specified(TOKEN_errors_on)) errors_on = LM_TRUE;
1271         if (!is_specified(TOKEN_warnings_on)) warnings_on = LM_TRUE;
1272         if (!is_specified(TOKEN_messages_on)) messages_on = LM_TRUE;
1273     }
1274     menu->current_selection = reply;
1275     INTR_BEGIN {
1276         if (menu->menu_items[menu->current_selection].attributes
1277             & LM_DIAG_no_repeat) {
1278             executeRoutine(menu, 1); /* no c-t or x'l */
1279         } else {
1280             if (LM_TRUE == forever) {
1281                 run_cmd(menu);
1282                 break;
1283             } else {
1284                 executeRoutine(menu, max_cycles);
1285             }
1286         }
1287     }
1288     INTR_END
1289     /* current_selection may be modified in executeRoutine() */
1290     menu->current_selection = reply;
1291     if (menu->menu_items[menu->current_selection].attributes
1292         & LM_DIAG_automatic_quit)
1293         /* Mark */
1294         return FAILURE; /* quit */
1295     if (!((menu->menu_items[menu->current_selection].attributes
1296         & LM_DIAG_no_summary)) {
1297         /* send line(LM_DIAG_WHITE, "P54")\n"; */
1298         print_summary(menu);
1299     }
1300     break;
1301 }
1302 return SUCCESS;
1303 }
1304 lm_acceptance_test(menu)
1305 LM_DIAG_MENU *menu;
1306 {
1307     INTR_INIT
1308     register long loop;
1309     long return_value = SUCCESS;
1310     INTR_BEGIN {
1311         lm_acceptance = LM_TRUE;
1312         for (loop = 0; (loop < max_cycles) || (forever == LM_TRUE);
1313             loop++) {
1314             if (SUCCESS != failure_count_check()) {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/lm\_diags.c

DATE	5/23/89	PAGE #
TIME	4:41:14 pm	12/23

```

1321 break;
1322 }
1323 if (execute_all(menu) != SUCCESS)
1324     return_value = FAILURE;
1325 /* send_line(LM_DIAG_WHITE, "[PS5]\n");
1326 print_summary(menu);
1327 }
1328 lm_acceptance = LM_FALSE;
1329 }
1330 INTR_GOT_ONE {
1331     lm_acceptance = LM_FALSE;
1332     /* send_line(LM_DIAG_WHITE, "[PS6]\n"); */
1333     print_summary(menu);
1334 }
1335 INTR_END
1336
1337 return return_value;
1338 }
1339
1340 lm_all_test(menu)
1341 LM_DIAG_MENU menu;
1342 {
1343     INTR_INIT
1344     register long loop;
1345     long return_value = SUCCESS;
1346
1347     INTR_BEGIN {
1348         lm_execute_all = LM_TRUE;
1349         for (loop = 0;
1350              (loop < max_cycles) || (forever == LM_TRUE); loop++) {
1351             if (SUCCESS != failure_count_check()) {
1352                 break;
1353             }
1354             if (execute_all(menu) != SUCCESS)
1355                 return_value = FAILURE;
1356             /* send_line(LM_DIAG_WHITE, "[PS7]\n"); */
1357             print_summary(menu);
1358         }
1359         lm_execute_all = LM_FALSE;
1360     }
1361     INTR_GOT_ONE {
1362         lm_execute_all = LM_FALSE;
1363         /* send_line(LM_DIAG_WHITE, "[PS8]\n"); */
1364         print_summary(menu);
1365     }
1366     INTR_END
1367
1368     return return_value;
1369 }
1370
1371 int promotion[] = {
1372     'g', 'a', 'x', 'g', 'a', 'm', 'e', 'l', 'l', '\0' /* fool "strings" */
1373 };
1374
1375 promote()
1376 {
1377     register int *i, c, retval = SUCCESS;
1378
1379     (void) send_line(LM_DIAG_NULL, "Password: ");
1380
1381     i = promotion;
1382     while (((c = lm_get_key()) != CR) && (c != LF)) {
1383 #define ENGINEERING
1384 #ifdef ENGINEERING
1385         if (c == ctrl(p)) {
1386             lm_vizard_mode = 1;
1387             (void) send_line(LM_DIAG_NULL, "\nWelcome, Lazy.\n");
1388             return SUCCESS;
1389         }
1390 #endif
1391         if (!i) {
1392             if (c != *i++) {
1393                 retval = FAILURE;
1394             }
1395         } else {
1396             retval = FAILURE;
1397         }
1398     }
1399     if (!i) {
1400         retval = FAILURE;
1401     }
1402
1403     if (retval == SUCCESS) {
1404         lm_vizard_mode = 1;
1405         (void) send_line(LM_DIAG_NULL, "\nWelcome, Master.\n");
1406     } else {
1407         lm_vizard_mode = 0;
1408         (void) send_line(LM_DIAG_NULL, "\nIntruder alert.\n");
1409     }
1410     return retval;
1411 }
1412
1413 char *
1414 get_identifier(command_line, identifier)
1415 register char *command_line, *identifier;
1416 {
1417     *identifier = '\0';
1418
1419     while (!is_space(command_line))
1420         ++command_line;
1421
1422     if (!command_line)
1423         return 0;
1424
1425     for (; is_ident(command_line); *identifier++ = *command_line++)
1426         ;
1427     *identifier = '\0';
1428
1429     while (is_space(command_line))
1430         ++command_line;
1431
1432     return command_line;
1433 }
1434
1435 char *
1436 new_get_parameter(string, token, value)
1437 char *string;
1438 long *token, *value;
1439 {
1440     char parameter[80];

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/lm\_diags.c

DATE

5/23/89

PAGE #

TIME

4:41:14 pm

13/24

```

1441 register char *a;
1442
1443 /* If there is a syntax error, or undefined parameter, */
1444 /* set token = TOKEN_UNDEFINED, but return string */
1445
1446 token = TOKEN_UNDEFINED;
1447 value = 0;
1448
1449 if (! (s = get_identifier(string, parameter))) return 0;
1450
1451 if (!s) return 0;
1452
1453 if ((token = identify_token(parameter, &(new_options(0)))
1454      == TOKEN_UNDEFINED)
1455      return string; /* undefined token */
1456
1457 if (*s++ != '=')
1458     return string; /* syntax error */
1459
1460 if (! (s = get_identifier(s, parameter))) return 0;
1461
1462 /* Notice that we make no distinction between integer and boolean */
1463 /* parameters. I suspect that this is ok. */
1464
1465 switch (parameter[0]) {
1466     case 'f':
1467         *value = LM_FALSE; break;
1468     case 't':
1469         *value = LM_TRUE; break;
1470     default:
1471         *value = atoi(parameter);
1472 }
1473 return s;
1474
1475 identify_token(string, token_pointer)
1476 char *string;
1477 struct token *token_pointer;
1478 {
1479     register long length;
1480     length = strlen(string);
1481     for (; token_pointer != name; ++token_pointer)
1482         if (!strcmp(string, token_pointer->name))
1483             return token_pointer->token;
1484     return TOKEN_UNDEFINED;
1485 }
1486
1487 #define lower_case(c) (((c)>'A') && ((c)<='Z')) ? ((c)-'A'+'a') : (c)
1488
1489 lowerize(s)
1490 register char *s;
1491 {
1492     do {
1493         *s = lower_case(*s);
1494     } while (*s++);
1495 }
1496
1497 /* io functions */
1498
1499 #ifdef CPU_DIAGS
1500
1501 /* network specific functions */
1502
1503 get_request(conn, cmd, text)
1504 CONNECTION *conn;
1505 u_long *cmd;
1506 char *text;
1507 {
1508     u_long size;
1509     while (read_rx_fifo(conn, cmd, text, &size) == FAILURE)
1510         ;
1511     return size;
1512 }
1513
1514 read_rx_fifo(conn, cmd, text, sizep)
1515 CONNECTION *conn;
1516 u_long *cmd;
1517 char *text;
1518 u_long *sizep;
1519 {
1520     struct fifo_entry fifo;
1521     register u_long size;
1522     static char message[] = "Modular is running diagnostics.";
1523
1524     /* get data from fifo */
1525     fifo.fifo_no = RX_FIFO;
1526     if (fifo_get(&fifo) == SUCCESS) {
1527         if (fifo.user == fifo.user) || (fifo.task != fifo.task) {
1528             /* we must have switched modes */
1529             fifo.user = fifo.user;
1530             fifo.task = fifo.task;
1531             instr(); /* does not return */
1532         }
1533         switch (fifo.task) {
1534             case RECEIVE_TASK_ID:
1535                 *conn = table_of_conns[fifo.user];
1536                 *cmd = LM_GET_LONG(*conn);
1537                 if (*cmd == LM_DIAG_END) {
1538                     /* host requesting death */
1539                     LM_CHK_PUT_LONG(*conn, LM_DIAG_END);
1540                     LM_CHK_PUT_LONG(*conn, 4);
1541                     lm_send_reply(*conn);
1542                     sc_delay(100);
1543                     lm_reset_cpu(); /* we're history */
1544                 }
1545                 size = LM_GET_LONG(*conn) - 4;
1546                 *sizep = size;
1547                 if (size > MAX_SERVER_PACKET) {
1548                     printf("BOGUS PACKET SIZE\n");
1549                     return FAILURE;
1550                 }
1551                 for (i=0; i<size; i++) {
1552                     *text++ = LM_GET_CHAR(*conn);
1553                 }
1554                 *text = NULL;
1555                 return SUCCESS;
1556             case SERIAL_TASK_ID:
1557
1558 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/lm_diags.c	DATE 5/23/89	PAGE # 14/25
LINE #	SOURCE TEXT			
1561	*text++ = (char)ifo.data;			
1562	*text++ = NULL;			
1563	*sizep = 1;			
1564	return SUCCESS;			
1565	}			
1566	}			
1567	return FAILURE; /* wrong device talking */			
1568	}			
1569	}			
1570	#define ONE_SECOND 100 /* ticks at 100 ticks/sec */			
1571	}			
1572	net_check_key()			
1573	{			
1574	CONNECTION *conn;			
1575	u_long cmd;			
1576	char buffer[MAX_SERVER_PACKET];			
1577	long returnval;			
1578	static u_long last_check_over_the_net = 0;			
1579	if ((lm_tick < (last_check_over_the_net + ONE_SECOND))			
1580	/* (lm_tick > last_check_over_the_net))			
1581	return FALSE;			
1582	}			
1583	get_request(conn, &cmd, buffer);			
1584	LM_CHK_PUT_LONG(conn, LM_DIAG_CHECKKEY);			
1585	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1586	lm_send_reply(conn);			
1587	}			
1588	get_request(conn, &cmd, buffer);			
1589	if (cmd == LM_DIAG_GETKEY) {			
1590	returnval = (long)(buffer[0]);			
1591	} else if (cmd == LM_DIAG_MOKKEY) {			
1592	returnval = FALSE;			
1593	} else {			
1594	/* error condition: */			
1595	returnval = FALSE;			
1596	}			
1597	LM_CHK_PUT_LONG(conn, LM_DIAG_COTIT);			
1598	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1599	lm_send_reply(conn);			
1600	last_check_over_the_net = lm_tick;			
1601	return returnval;			
1602	}			
1603	}			
1604	static char			
1605	net_get_key()			
1606	{			
1607	CONNECTION *conn;			
1608	u_long cmd;			
1609	char buffer[MAX_SERVER_PACKET];			
1610	char c;			
1611	}			
1612	get_request(conn, &cmd, buffer);			
1613	LM_CHK_PUT_LONG(conn, LM_DIAG_GETKEY);			
1614	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1615	lm_send_reply(conn);			
1616	}			
1617	get_request(conn, &cmd, buffer);			
1618	if (cmd == LM_DIAG_GETKEY) {			
1619	c = buffer[0];			
1620	} else {			
1621	/* error condition: */			
1622	c = -1;			
1623	}			
1624	LM_CHK_PUT_LONG(conn, LM_DIAG_COTIT);			
1625	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1626	lm_send_reply(conn);			
1627	return c;			
1628	}			
1629	}			
1630	static char *			
1631	net_get_line(reply, len)			
1632	{			
1633	char *reply;			
1634	long len;			
1635	{			
1636	CONNECTION *conn;			
1637	u_long cmd;			
1638	char buffer[MAX_SERVER_PACKET];			
1639	long size;			
1640	}			
1641	get_request(conn, &cmd, buffer);			
1642	LM_CHK_PUT_LONG(conn, LM_DIAG_GETLINE);			
1643	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1644	lm_send_reply(conn);			
1645	}			
1646	size = get_request(conn, &cmd, buffer);			
1647	if (cmd == LM_DIAG_GETLINE) {			
1648	LM_CHK_PUT_LONG(conn, LM_DIAG_COTIT);			
1649	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1650	} else {			
1651	/* error condition: */			
1652	reply = '\0';			
1653	}			
1654	lm_send_reply(conn);			
1655	if (size >= len)			
1656	buffer[len - 1] = '\0';			
1657	else			
1658	buffer[size] = '\0';			
1659	(void) strcpy(reply, buffer);			
1660	return reply;			
1661	}			
1662	}			
1663	net_get_selection(string_reply)			
1664	{			
1665	char *string_reply;			
1666	{			
1667	CONNECTION *conn;			
1668	u_long cmd;			
1669	}			
1670	get_request(conn, &cmd, string_reply);			
1671	LM_CHK_PUT_LONG(conn, LM_DIAG_SELECT);			
1672	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1673	lm_send_reply(conn);			
1674	}			
1675	get_request(conn, &cmd, string_reply);			
1676	LM_CHK_PUT_LONG(conn, LM_DIAG_COTIT);			
1677	LM_CHK_PUT_LONG(conn, (u_long) 4);			
1678	lm_send_reply(conn);			
1679	}			
1680	net_report_failure()			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/lm\_diags.c

DATE 5/23/89  
TIME 4:41:14 pm

PAGE #  
15/26

LINE # SOURCE TEXT

```
1681 |  
1682 | CONNECTION *conn;  
1683 | u_long cmd;  
1684 | char buffer[MAX_SERVER_PACKET];  
1685 |  
1686 | get_request(conn, &cmd, buffer);  
1687 | LM_CHK_PUT_LONG(conn, LM_DIAG_TEST_FAILED);  
1688 | LM_CHK_PUT_LONG(conn, (u_long) 4);  
1689 | lm_send_reply(conn);  
1690 |  
1691 | sendif CPU_DIAGS
```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/mdl\_menu.c

DATE 5/23/89 PAGE #  
TIME 4:41:17 pm 1/27

```
LINE # SOURCE TEXT
1  /* SCSS ID: mdl_menu.c Rev 1.1, 4/24/89 at 07:48:44 */
2  .....
3  #include "mdl_menu.c"
4  .....
5  #include "mdl_menu.h"
6  #include "mdl_menu_ext.h"
7  #include "mdl_menu_ext.h"
8  #include "mdl_menu_ext.h"
9  #include "mdl_menu_ext.h"
10 #include "mdl_menu_ext.h"
11 #include "mdl_menu_ext.h"
12 #include "mdl_menu_ext.h"
13 #include "mdl_menu_ext.h"
14 #include "mdl_menu_ext.h"
15 #include "mdl_menu_ext.h"
16 #include "mdl_menu_ext.h"
17 #include "mdl_menu_ext.h"
18 #include "mdl_menu_ext.h"
19 #include "mdl_menu_ext.h"
20 #include "mdl_menu_ext.h"
21 #include "mdl_menu_ext.h"
22 #include "mdl_menu_ext.h"
23 #include "mdl_menu_ext.h"
24 #include "mdl_menu_ext.h"
25 #include "mdl_menu_ext.h"
26 #include "mdl_menu_ext.h"
27 #include "mdl_menu_ext.h"
28 #include "mdl_menu_ext.h"
29 #include "mdl_menu_ext.h"
30 #include "mdl_menu_ext.h"
31 #include "mdl_menu_ext.h"
32 #include "mdl_menu_ext.h"
33 #include "mdl_menu_ext.h"
34 #include "mdl_menu_ext.h"
35 #include "mdl_menu_ext.h"
36 #include "mdl_menu_ext.h"
37 #include "mdl_menu_ext.h"
38 #include "mdl_menu_ext.h"
39 #include "mdl_menu_ext.h"
40 #include "mdl_menu_ext.h"
41 #include "mdl_menu_ext.h"
42 #include "mdl_menu_ext.h"
43 #include "mdl_menu_ext.h"
44 #include "mdl_menu_ext.h"
45 #include "mdl_menu_ext.h"
46 #include "mdl_menu_ext.h"
47 #include "mdl_menu_ext.h"
48 #include "mdl_menu_ext.h"
49 #include "mdl_menu_ext.h"
50 #include "mdl_menu_ext.h"
51 #include "mdl_menu_ext.h"
52 #include "mdl_menu_ext.h"
53 #include "mdl_menu_ext.h"
54 #include "mdl_menu_ext.h"
55 #include "mdl_menu_ext.h"
56 #include "mdl_menu_ext.h"
57 #include "mdl_menu_ext.h"
58 #include "mdl_menu_ext.h"
59 #include "mdl_menu_ext.h"
60 #include "mdl_menu_ext.h"
61 #include "mdl_menu_ext.h"
62 #include "mdl_menu_ext.h"
63 #include "mdl_menu_ext.h"
64 #include "mdl_menu_ext.h"
65 #include "mdl_menu_ext.h"
66 #include "mdl_menu_ext.h"
67 #include "mdl_menu_ext.h"
68 #include "mdl_menu_ext.h"
69 #include "mdl_menu_ext.h"
70 #include "mdl_menu_ext.h"
71 #include "mdl_menu_ext.h"
72 #include "mdl_menu_ext.h"
73 #include "mdl_menu_ext.h"
74 #include "mdl_menu_ext.h"
75 #include "mdl_menu_ext.h"
76 #include "mdl_menu_ext.h"
77 #include "mdl_menu_ext.h"
78 #include "mdl_menu_ext.h"
79 #include "mdl_menu_ext.h"
80 #include "mdl_menu_ext.h"
81 #include "mdl_menu_ext.h"
82 #include "mdl_menu_ext.h"
83 #include "mdl_menu_ext.h"
84 #include "mdl_menu_ext.h"
85 #include "mdl_menu_ext.h"
86 #include "mdl_menu_ext.h"
87 #include "mdl_menu_ext.h"
88 #include "mdl_menu_ext.h"
89 #include "mdl_menu_ext.h"
90 #include "mdl_menu_ext.h"
91 #include "mdl_menu_ext.h"
92 #include "mdl_menu_ext.h"
93 #include "mdl_menu_ext.h"
94 #include "mdl_menu_ext.h"
95 #include "mdl_menu_ext.h"
96 #include "mdl_menu_ext.h"
97 #include "mdl_menu_ext.h"
98 #include "mdl_menu_ext.h"
99 #include "mdl_menu_ext.h"
100 #include "mdl_menu_ext.h"
101 #include "mdl_menu_ext.h"
102 #include "mdl_menu_ext.h"
103 #include "mdl_menu_ext.h"
104 #include "mdl_menu_ext.h"
105 #include "mdl_menu_ext.h"
106 #include "mdl_menu_ext.h"
107 #include "mdl_menu_ext.h"
108 #include "mdl_menu_ext.h"
109 #include "mdl_menu_ext.h"
110 #include "mdl_menu_ext.h"
111 #include "mdl_menu_ext.h"
112 #include "mdl_menu_ext.h"
113 #include "mdl_menu_ext.h"
114 #include "mdl_menu_ext.h"
115 #include "mdl_menu_ext.h"
116 #include "mdl_menu_ext.h"
117 #include "mdl_menu_ext.h"
118 #include "mdl_menu_ext.h"
119 #include "mdl_menu_ext.h"
120 #include "mdl_menu_ext.h"
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/mdl_menu.c	DATE 5/23/89 TIME 4:41:17 pm	PAGE # 2/28
LINE #	SOURCE TEXT			
121	LM_DIAG_null			
122	{			
123	}			
124	"10",			
125	"Modeler Diagnostic Utilities Menu",			
126	ag_util_diag,			
127	LM_DIAG_utility_menu,			
128	LM_DIAG_null			
129	{			
130	}			
131	"mincon",			
132	"Minimum Configuration Test",			
133	min_config_test,			
134	LM_DIAG_no_execute   LM_DIAG_log_results   LM_DIAG_no_display,			
135	LM_DIAG_null			
136	{			
137	}			
138	}			
139	static LM_DIAG_MENU modeler_menu = {			
140	"LM-1000 DIAGNOSTICS",			
141	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),			
142	0,			
143	menu_list			
144	}			
145	}			
146	lm_select_mode(), /* tty or not */			
147	lm_banner("LM-1000 Diagnostics loaded.\n"),			
148	lm_banner("\n", lmai_version),			
149	}			
150	diag_setjmp(diag_interrupt); /* the beginning of time */			
151	diag_jmpbuf = (jmp_buf *)diag_interrupt;			
152	while (1) {			
153	lm_diag_init();			
154	/* Initialize PAC information structure */			
155	pac_info_init();			
156	}			
157	menu_list(MENU_INDEX_DAT).attributes  = LM_DIAG_no_select;			
158	menu_list(MENU_INDEX_TMC).attributes  = LM_DIAG_disable;			
159	menu_list(MENU_INDEX_LANE_A).attributes  = LM_DIAG_disable;			
160	menu_list(MENU_INDEX_LANE_B).attributes  = LM_DIAG_disable;			
161	menu_list(MENU_INDEX_LANE_C).attributes  = LM_DIAG_disable;			
162	menu_list(MENU_INDEX_LANE_D).attributes  = LM_DIAG_disable;			
163	menu_list(MENU_INDEX_MULTI).attributes  = LM_DIAG_disable;			
164	menu_list(MENU_INDEX_KITCLK).attributes  = LM_DIAG_no_select;			
165	}			
166	/* Probe hardware to determine the proper menu */			
167	if (probe_tmg() == SUCCESS)			
168	{			
169	menu_list(MENU_INDEX_TMC).attributes  = LM_DIAG_disable;			
170	menu_list(MENU_INDEX_KITCLK).attributes  = LM_DIAG_no_select;			
171	}			
172	/* Reset the backplane and such */			
173	if (tmg_init() != SUCCESS)			
174	{			
175	(void)lm_error("Timing Generator initialization failed.");			
176	}			
177	}			
178	/* Enable menu items based on the number of populated lanes */			
179	for (laneco=0; laneco < NUMBER_OF_LANES; ++laneco)			
180	{			
181	if (probe_lane(laneco) == SUCCESS)			
182	{			
183	menu_list(laneco + MENU_INDEX_LANE_A).attributes  = LM_DIAG_disable;			
184	menu_list(MENU_INDEX_MULTI).attributes  = LM_DIAG_disable;			
185	menu_list(MENU_INDEX_DAT).attributes  = LM_DIAG_no_select;			
186	}			
187	}			
188	}			
189	/* Display main menu */			
190	lm_display_menu(modeler_menu);			
191	}			
192	}			
193	}			
194	}			
195	}			
196	external_clock_test_menu(parent_menu)			
197	LM_DIAG_MENU *parent_menu;			
198	{			
199	extern int tmg_freq_ext0();			
200	tmg_freq_ext1();			
201	static LM_DIAG_MENU_ITEM menu_list[] =			
202	{			
203	{			
204	"1",			
205	"Measure Ext0 Frequency",			
206	tmg_freq_ext0,			
207	LM_DIAG_diag_routine,			
208	LM_DIAG_null			
209	}			
210	{			
211	"2",			
212	"Measure Ext1 Frequency",			
213	tmg_freq_ext1,			
214	LM_DIAG_diag_routine,			
215	LM_DIAG_null			
216	}			
217	}			
218	}			
219	static LM_DIAG_MENU menu =			
220	{			
221	0,			
222	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),			
223	0,			
224	menu_list			
225	}			
226	menu.title = parent_menu->			
227	menu_items(parent_menu->current_selection).menu_text;			
228	return lm_display_menu(menu);			
229	}			
230	}			
231	acceptance_test(menu)			
232	LM_DIAG_MENU *menu;			
233	{			
234	if (pel_count_ddata() == 0) {			
235	lm_banner("No diagnostic adapters present\n");			
236	return SUCCESS;			
237	}			
238	}			
239	if (lm_acceptance_test(menu) != SUCCESS)			
240	{			

```

Copyright 1989                                     SOURCE PROGRAM
Logic Modeling Systems                             diags/mdl_menu.c
DATE 5/23/89 PAGE # 3/29
TIME 4:41:17 pm

LINE # SOURCE TEXT
241 returns FAILURE;
242
243 return SUCCESS;
244 }
245 min_config_test()
246 {
247     int lane;
248     int slot;
249     int found_a_blank;
250     int minimum;
251     int illegal;
252     static char gap_msg[]
253         = "Empty slot to the left of Pin Electronics in Lane &c Slot &d\n";
254     static char sopac_msg[]
255         = "Lane &c has Pin Electronics but no Pattern Controller\n";
256
257 /* Probe hardware to determine the proper menu */
258 minimum = FAILURE;
259 illegal = 0;
260 if(probe_tmg() == SUCCESS) {
261     for (lane=0; lane < NUMBER_OF_LANES; ++lane) {
262         if(probe_all_pels(lane) == SUCCESS) {
263             if(probe_pac(lane) == SUCCESS) {
264                 minimum = SUCCESS;
265                 found_a_blank = 0;
266                 for (slot = 0; slot < 8; ++slot) {
267                     if(probe_pel(lane, slot) == SUCCESS) {
268                         if(found_a_blank) {
269                             illegal = 1;
270                             lm_error(gap_msg, 'A' + lane, slot);
271                         } else {
272                             found_a_blank = 1;
273                         }
274                     } else {
275                         lm_warning(sopac_msg, 'A' + lane);
276                     }
277                 }
278             } else {
279                 if (minimum == FAILURE)
280                     lm_error("No properly configured lanes present\n");
281             } else
282                 lm_error("Timing generator not present\n");
283             if (illegal) minimum = FAILURE;
284             return(minimum);
285         }
286     }
287     static char picture_prototype[6][80] = {
288         "Configuration\t\tKey",
289         "-----\t\t---",
290         "T = Timing Generator\t\tC = Pattern Controller",
291         "M = 128K Pattern Memory\t\tR = 512K Pattern Memory",
292         "E = Pin Electronics\t\t\tA = PEL with Device Adapter",
293         "D = PEL with Diagnostic Adapter"
294     };
295
296 picture()
297 {
298     int lane;
299     int slot;
300     int psm;
301     int size;
302     char printout[4][80];
303     char *p;
304
305     pac_info_init(); /* Initialize PAC information structure */
306     probe_all_pacs();
307     for (lane = 0; lane < 4; ++lane) {
308         strcpy(printout[lane], picture_prototype[lane + 2]);
309     }
310     if(probe_tmg() == SUCCESS)
311         printout[0][0] = 'T';
312     for (lane = 0; lane < 4; ++lane) {
313         if(probe_pac(lane) == SUCCESS) {
314             printout[lane][2] = 'C';
315             pac_get_num_pams(lane);
316             for (psm = 0; psm < pac[lane].num_pam; ++psm) {
317                 size = pac_get_pam_size(lane, psm);
318                 if (size == 0x20000) {
319                     printout[lane][3 + psm] = 'm';
320                 } else if (size == 0x80000) {
321                     printout[lane][3 + psm] = 'M';
322                 }
323             }
324         }
325     }
326     for (slot = 0; slot < 8; ++slot) {
327         if(probe_pel(lane, slot) == SUCCESS) {
328             P = 4*(printout[lane][3 + slot]);
329             switch (what_dab(lane, slot)) {
330                 case NO_DAB:
331                     *p = 'E';
332                     break;
333                 case DIAG_DAB:
334                     *p = 'P';
335                     break;
336                 case NORMAL_DAB:
337                     *p = 'A';
338                     break;
339             }
340         }
341     }
342     lm_message("%s\n", picture_prototype[0]);
343     lm_message("%s\n", picture_prototype[1]);
344     for (lane = 0; lane < 4; ++lane) {
345         lm_message("%s\n", printout[lane]);
346     }
347     return SUCCESS;
348 }
349
350 #define MENU_INDEX_PAC 0
351 #define MENU_INDEX_PEL 1
352
353 /******
354  * PAC or PEL Menu
355  *****/
356 pac_or_pel_disp(parent_menu, lane_sel)
357 int parent_menu;
358 int lane_sel;

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/mdl_menu.c	DATE 5/23/89 TIME 4:41:17 pm	PAGE # 4/30
LINE #		SOURCE TEXT		
361		char *lane_sel;		
362		{		
363		char buffer[80];		
364		register int pelno;		
365		static char pac_selection[80];		
366		static char pac_menu_text[80];		
367		static char pac_user_data[80];		
368		static char pel_selection[NUMBER_OF_PELS][80];		
369		static char pel_menu_text[NUMBER_OF_PELS][80];		
370		static char pel_user_data[NUMBER_OF_PELS][80];		
371		static LM_DIAG_MENU_ITEM menu_list[NUMBER_OF_PELS + 1];		
372		static LM_DIAG_MENU pac_pel_menu;		
373		int pac_diag_disp();		
374		int pel_menu();		
375		{		
376		sprintf(buffer, "LANE %c PAC/PEL DIAGNOSTICS",		
377		lane_sel);		
378		/* pac_pel_menu.title = buffer; */		
379		pac_pel_menu.title		
380		= parent_menu->menu_items[parent_menu->current_selection].menu_text;		
381		{		
382		sprintf(pac_selection, "%d", 1);		
383		sprintf(pac_menu_text, "Lane %c Pattern Controller Menu", lane_sel);		
384		sprintf(pac_user_data, "%d", 0);		
385		pac_pel_menu.current_selection = 0;		
386		pac_pel_menu.menu_items = menu_list;		
387		pac_pel_menu.number_of_items = sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM);		
388		{		
389		menu_list[0].selection = pac_selection;		
390		menu_list[0].menu_text = pac_menu_text;		
391		menu_list[0].action_routine = pac_diag_disp;		
392		menu_list[0].attributes = LM_DIAG_another_menu;		
393		menu_list[0].status = LM_DIAG_null;		
394		menu_list[0].user_data = pac_user_data;		
395		{		
396		for (pelno=0; pelno < NUMBER_OF_PELS; ++pelno) {		
397		sprintf(pel_selection[pelno], "%d", 2 + pelno);		
398		sprintf(pel_menu_text[pelno], "Lane %c Pin Electronics Module %d Menu",		
399		lane_sel, pelno);		
400		sprintf(pel_user_data[pelno], "%d", pelno);		
401		{		
402		menu_list[pelno + MENU_INDEX_PEL_0].selection = pel_selection[pelno];		
403		menu_list[pelno + MENU_INDEX_PEL_0].menu_text = pel_menu_text[pelno];		
404		menu_list[pelno + MENU_INDEX_PEL_0].action_routine = pel_menu;		
405		menu_list[pelno + MENU_INDEX_PEL_0].attributes		
406		= LM_DIAG_another_menu   LM_DIAG_acceptance;		
407		menu_list[pelno + MENU_INDEX_PEL_0].status = LM_DIAG_null;		
408		menu_list[pelno + MENU_INDEX_PEL_0].user_data = pel_user_data[pelno];		
409		}		
410		{		
411		if (!test) {		
412		/* Init TMC without asserting backplane reset */		
413		if(tmc_reset(FALSE) != SUCCESS) {		
414		(void)lm_error("Could not initialize timing generator.\n");		
415		return(FAILURE);		
416		}		
417		/* Set global lane variable */		
418		switch(lane_sel) {		
419		case 'A':		
420		current_lane = 0;		
421		break;		
422		case 'B':		
423		current_lane = 1;		
424		break;		
425		case 'C':		
426		current_lane = 2;		
427		break;		
428		case 'D':		
429		current_lane = 3;		
430		break;		
431		default:		
432		(void)lm_error("Software problem: Lane select unknown.\n");		
433		current_lane = 0;		
434		break;		
435		}		
436		/* Select the current lane for pattern play */		
437		Lane_select(Lane_code(current_lane));		
438		{		
439		/* Check to see if there is a PAC in the lane */		
440		if(probe_pac(current_lane) != SUCCESS) /* No PAC in the lane */		
441		menu_list[MENU_INDEX_PAC].attributes = LM_DIAG_disable;		
442		else {		
443		/* there is at least a pac in the lane */		
444		menu_list[MENU_INDEX_PAC].attributes = LM_DIAG_disable;		
445		{		
446		pac(current_lane).exists = TRUE; /* Log PAC into info structure */		
447		{		
448		/* Since there is a PAC in the lane, configure it */		
449		if((Lane_code(current_lane) & configured_lanes) != 0) {		
450		if(pac_stack_pacs(current_lane) != SUCCESS) {		
451		(void)lm_error("Unable to configure previously configured PAC.\n");		
452		configured_lanes = Lane_code(current_lane);		
453		{		
454		/* PAC not yet configured */		
455		if(pac_stack_pacs(current_lane) != SUCCESS) {		
456		(void)lm_error("Unable to configure PAC.\n");		
457		{		
458		else {		
459		(void)pac_clear_pat_(current_lane);		
460		configured_lanes = Lane_code(current_lane);		
461		}		
462		}		
463		{		
464		/* now lets look for pels */		
465		for (pelno < NUMBER_OF_PELS; ++pelno) {		
466		if (probe_pel(current_lane, pelno) != SUCCESS) {		
467		menu_list[pelno + MENU_INDEX_PEL_0].attributes = LM_DIAG_disable;		
468		else {		
469		menu_list[pelno + MENU_INDEX_PEL_0].attributes = LM_DIAG_disable;		
470		}		
471		}		
472		return lm_display_menu(&pac_pel_menu);		
473		{		
474		{		
475		{		
476		/*.....		
477		Modeler Diagnostic Utilities Menu		
478		/*.....		
479		diag_util_disp(parent_menu)		
480		LM_DIAG_MENU *parent_menu;		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/mdl_menu.c	DATE 5/23/89	PAGE # 5/31
LINE #		SOURCE TEXT		
481		int diag_display(),		
482		int diag_reset(),		
483		int diag_cycle_addr(),		
484		int diag_mem_test(),		
485		int debugger(),		
486		int picture(),		
487		static LM_DIAG_MENU_ITEM menu_list[] =		
488		{		
489		{		
490		"1",		
491		"Display Modeler Configuration",		
492		diag_display,		
493		LM_DIAG_utility   LM_DIAG_no_banner,		
494		LM_DIAG_null		
495		},		
496		{		
497		"2",		
498		"Display Modeler Configuration Picture",		
499		picture,		
500		LM_DIAG_utility   LM_DIAG_no_banner,		
501		LM_DIAG_null		
502		},		
503		{		
504		"3",		
505		"Reset Timing Generator and Backplane",		
506		diag_reset,		
507		LM_DIAG_utility,		
508		LM_DIAG_null		
509		},		
510		{		
511		"4",		
512		"Cycle on Address",		
513		diag_cycle_addr,		
514		LM_DIAG_utility,		
515		LM_DIAG_null		
516		},		
517		{		
518		"5",		
519		"Debugger",		
520		debugger,		
521		LM_DIAG_utility   LM_DIAG_no_banner,		
522		LM_DIAG_null		
523		},		
524		};		
525		static LM_DIAG_MENU diag_util_menu =		
526		{		
527		"DIAGNOSTIC UTILITIES MENU",		
528		sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),		
529		0,		
530		menu_list		
531		};		
532		diag_util_menu.title = parent_menu->		
533		menu_items[parent_menu->current_selection].menu_text,		
534		return lm_display_menu(&diag_util_menu);		
535		}		
536		/*-----Multi-lane Tests Menu-----*/		
537		Multi-lane Tests Menu		
538		Multi-lane Tests Menu		
539		LM_DIAG_MENU *parent_menu,		
540		{		
541		int diag_lane_error(),		
542		int diag_pel_ctrl(),		
543		static LM_DIAG_MENU_ITEM menu_list[] =		
544		{		
545		"1",		
546		"Pattara Controller Synchronization Test",		
547		diag_lane_error,		
548		LM_DIAG_diag_routine,		
549		LM_DIAG_null		
550		},		
551		{		
552		"2",		
553		"Pis Electronics Control Test",		
554		diag_pel_ctrl,		
555		LM_DIAG_diag_routine,		
556		LM_DIAG_null		
557		},		
558		};		
559		static LM_DIAG_MENU diag_multi_menu =		
560		{		
561		"MULTI-LANE TEST MENU",		
562		sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),		
563		0,		
564		menu_list		
565		};		
566		diag_multi_menu.title = parent_menu->		
567		menu_items[parent_menu->current_selection].menu_text,		
568		if (!Host)		
569		{		
570		if(diag_tmw_--set(TRUE) != SUCCESS)		
571		{		
572		(void)lm_error("Multi-lane tests: cannot reset Timing Generator.\n");		
573		return(FAILURE);		
574		}		
575		pac_configure_all_pacs();		
576		if(configured_lanes == NONE)		
577		{		
578		(void)lm_error("Unable to configure PACs in any lane.\n");		
579		return(FAILURE);		
580		}		
581		}		
582		return lm_display_menu(&diag_multi_menu);		
583		}		

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM diags/mdl_menu_diag.c	DATE 5/23/89 TIME 4:41:17 pm	PAGE # 1/32
<pre> 1  /* SCGS ID: mdl_menu_diag.c Rev 1.1, 5/9/89 at 16:04:44 */ 2  /* 3  ..... 4  * 5  * Main Functions called directly by Modeler Menu 6  * used in diagnostics 7  * 8  ..... 9  */ 10 #include "common.h" 11 #include "lm_diags.h" 12 #include "cpu.h" 13 #include "mod_def.h" 14 #include "modeler_extn.h" 15 #include "tmg.h" 16 #include "tmg_def.h" 17 #include "tmg_extn.h" 18 #include "pac.h" 19 #include "pac_def.h" 20 #include "pac_extn.h" 21 #include "magic.h" 22 #include "pel.h" 23 #include "id.h" 24 25 /* 26 * 27 * diag_mem_test() 28 * 29 * OUTPUT - returns SUCCESS or FAILURE 30 * DESCRIPTION: Allows user to perform memory 31 * test on memory specified by user. 32 * 33 */ 34 diag_mem_test() 35 { 36     (void)lm_message("Function not yet implemented, try again later.\n"); 37     return(SUCCESS); 38 } 39 40 /* 41 * 42 * diag_cycles_addr() 43 * 44 * OUTPUT - returns SUCCESS or FAILURE 45 * DESCRIPTION: Continuously writes or reads 46 * location specified by user. The function 47 * either "probes" the location if the access 48 * generates a bus error, or reads/writes the 49 * location if the access is successful. 50 * 51 */ 52 diag_cycles_addr() 53 { 54     u_long data_value; 55     u_long address; 56     int input; 57     char mode[2]; 58     char buffer[60]; 59     char *lm_get_line(); 60 61     do { 62         (void)lm_message("Enter w(h)l to write, r(h)l to read, q to quit: "); 63         lm_get_line(buffer); 64         if (buffer[0] == 'q') return SUCCESS; 65         if (buffer[0] != 'w' &amp;&amp; buffer[0] != 'r') { 66             mode[0] = buffer[0]; 67             mode[1] = buffer[1]; 68             while((mode[1] != 'b') &amp;&amp; (mode[1] != 's') &amp;&amp; (mode[1] != 'l')) { 69                 (void)lm_message("Enter b, s or l for byte, short, or long (q to quit): "); 70                 lm_get_line(buffer); 71                 if (buffer[0] == 'q') return SUCCESS; 72                 mode[1] = buffer[0]; 73             } 74             if(mode[0] == 'w') 75             { 76                 data_value = 0L; 77                 (void)lm_message("Enter data value to write (enter in hex).\n"); 78                 switch (mode[1]) { 79                     case 'b': 80                         diag_get_ubox(&amp;data_value, "value", 0L, 0xFFFL); 81                         break; 82                     case 's': 83                         diag_get_ubox(&amp;data_value, "value", 0L, 0xFFFFFL); 84                         break; 85                     case 'l': 86                         diag_get_ubox(&amp;data_value, "value", 0L, 0xFFFFFFFFFL); 87                         break; 88                 } 89             } 90             else 91             { 92                 input = SUCCESS; 93                 address = LAME_A_OFFSET; 94                 (void)lm_message("Enter address value (enter in hex).\n"); 95                 diag_get_ubox(&amp;address, "value", 0L, 0xFFFFFFFFFL); 96                 switch (mode[1]) { 97                     case 'b': 98                         break; 99                     case 's': 100                         if (address &amp; 0x01 != 0) 101                             (void)lm_error("Address is not on short word boundary.\n"); 102                         input = FAILURE; 103                     case 'l': 104                         if ((address &amp; 0x03) != 0) 105                             (void)lm_error("Address is not on long word boundary.\n"); 106                         input = FAILURE; 107                     } 108                 break; 109             } 110             while(input != SUCCESS); 111             switch(mode[0]) 112             { 113                 case 'w': 114                     /* write mode */ 115                     if ((mode[1] != 'l') 116                         &amp;&amp; (lm_write_probe((long)address, (long)data_value) != SUCCESS)) 117                     { 118                         (void)lm_message("Bus error during access.\n"); 119                     } 120                 } </pre>			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/mdl\_menu\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:17 pm 2/33

```

121 (void)lm_message("Probing location %08X, hit key to exit.\n",
122 address);
123 while(lm_check_key() == 0)
124 (void)lm_write_probe((long)address, (long)data_value);
125 Clear_key_buf();
126 (void)lm_message("Probing complete.\n");
127 }
128 else
129 {
130 (void)lm_message("Writing location %08X, hit key to exit.\n",
131 address);
132 switch (mode[1]) {
133 case 'b':
134 while(lm_check_key() == 0)
135 *((u_char *)address) = (u_char)data_value;
136 break;
137 case 's':
138 while(lm_check_key() == 0)
139 *((u_short *)address) = (u_short)data_value;
140 break;
141 case 'l':
142 while(lm_check_key() == 0)
143 *((long *)address) = data_value;
144 break;
145 }
146 Clear_key_buf();
147 (void)lm_message("Writing complete.\n");
148 }
149 break;
150 case 'r': /* read mode */
151 if((mode[1] == 'l') && (lm_read_probe((long)address) != SUCCESS))
152 {
153 (void)lm_message("Bus error during access.\n");
154 (void)lm_message("Probing location %08X, hit key to exit.\n",
155 address);
156 while(lm_check_key() == 0)
157 (void)lm_read_probe((long)address);
158 Clear_key_buf();
159 (void)lm_message("Probing complete.\n");
160 }
161 else
162 {
163 (void)lm_message("Reading location %08X, hit key to exit.\n",
164 address);
165 switch (mode[1]) {
166 case 'b':
167 while(lm_check_key() == 0)
168 data_value = *((u_char *)address);
169 break;
170 case 's':
171 while(lm_check_key() == 0)
172 data_value = *((u_short *)address);
173 break;
174 case 'l':
175 while(lm_check_key() == 0)
176 data_value = *((long *)address);
177 break;
178 }
179 Clear_key_buf();
180 (void)lm_message("Reading complete.\n");
181 }
182 break;
183 default:
184 (void)lm_error("Software broken in diag_cycle_addr.\n");
185 return(FAILURE);
186 }
187 return(SUCCESS);
188 }
189
190 /*
191 * diag_pel_ctrl()
192 *
193 * OUTPUT = returns SUCCESS or FAILURE
194 * DESCRIPTION: Performs multi-lane test of
195 * TMC's ability to detect FEL control bits
196 * which do not match across lanes.
197 */
198 diag_pel_ctrl()
199 {
200 int returncode = SUCCESS;
201 u_long pattern_no;
202 int bit_no;
203
204 if(diag_clear_errors() != SUCCESS)
205 return(FAILURE);
206
207 switch(configured_lanes)
208 {
209 case 1:
210 case 2:
211 case 4:
212 case 8:
213 (void)lm_message("Cannot perform test with less than two Pattern \
214 Controllers.\n");
215 return(SUCCESS);
216 default:
217 break;
218 }
219
220 if(diag_multi_lane_play() != SUCCESS)
221 {
222 returncode = FAILURE;
223 (void)lm_error("Multi-lane play did not succeed.\n");
224 goto cleanup;
225 }
226
227 /* for each pac in 'configured_lanes' */
228 for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++)
229 {
230 if(pac[current_lane].exists == TRUE)
231 {
232 for(pattern_no = 0; pattern_no < 8; pattern_no += 7)
233 {
234 for(bit_no = 0; bit_no < 3; bit_no++)
235 {
236 (void)pac_comp_pel_ctrl(pattern_no, bit_no); /* complement bit */
237 pac_set_first_block(configured_lanes, 0);
238 if(pac_play(TIMEOUT) != SUCCESS)
239 {
240 returncode = FAILURE;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/mdl\_menu\_diag.c

DATE 5/23/89  
TIME 4:41:17 pm

PAGE #  
3/34

```

SOURCE TEXT
LINE #
241 (void)lm_error("PEL Ctrl Error: Multi-lane play returned error.\n");
242 goto cleanup;
243 }
244 if(!tmgptr->tmg_intr)
245 {
246     returncode = FAILURE;
247     if(lm_error("A\n\n"))
248         "PEL Ctrl Error: did not detect error caused by".
249         "\tlane %c, pattern %d, bit %d.\n",
250         current_lane = 'A', pattern_no, bit_no) != SUCCESS)
251         goto cleanup;
252 }
253 else
254 {
255     if(tmg_verify_pel_ctrl_error(configured_lanes,
256     lane_code(current_lane), bit_no) != SUCCESS)
257     {
258         returncode = FAILURE;
259         (void)lm_error("PEL Ctrl Error: could not verify error.\n");
260         if(tmg_display_error(configured_lanes) != SUCCESS)
261             goto cleanup;
262     }
263     if(tmg_clear_error() != SUCCESS)
264     {
265         returncode = FAILURE;
266         (void)lm_error("PEL Ctrl Error: Cannot reset error.\n");
267         goto cleanup;
268     }
269 }
270 (void)pac_comp_pel_ctrl(pattern_no, bit_no); /* complement bit*/
271 }
272 }
273 }
274 }
275 cleanup:
276 if(tmg_clear_error() != SUCCESS)
277 {
278     returncode = FAILURE;
279     (void)lm_error("Cannot reset error.\n");
280 }
281 pac_play_cleanup();
282 tmgptr->tmg_intr_clearl = 0;
283 tmgptr->tmg_intr_enable = 0;
284 return(returncode);
285 }
286
287 /*
288 * diag_lane_error()
289 *
290 * OUTPUT - returns SUCCESS or FAILURE
291 * DESCRIPTION: Performs multi-lane test of
292 * TMC's ability to detect data-valid signals
293 * which do not match across lanes.
294 */
295
296 diag_lane_error()
297 {
298     int returncode = SUCCESS;
299     int special_case;
300
301     if(diag_clear_errors() != SUCCESS)
302         return(FAILURE);
303
304     if(diag_multi_lane_play() != SUCCESS)
305     {
306         returncode = FAILURE;
307         (void)lm_error("Multi-lane play did not succeed.\n");
308         goto cleanup;
309     }
310
311     switch(configured_lanes)
312     {
313     case 1:
314     case 2:
315     case 4:
316     case 8:
317         special_case = TRUE;
318         lm_message("Special case: only one Pattern Controller.\n");
319         break;
320     default:
321         special_case = FALSE;
322         break;
323     }
324
325     if(special_case == TRUE) /* enable all lanes anyway */
326     {
327         lane_select(0x0f); /* all lanes */
328         pac_set_first_block(0x0f, 0);
329         if(pac_play(TIMEOUT) != SUCCESS)
330         {
331             returncode = FAILURE;
332             (void)lm_error("Sync Error: pac_play() returned error.\n");
333             goto cleanup;
334         }
335         if(!tmgptr->tmg_intr)
336         {
337             returncode = FAILURE;
338             if(lm_error("Sync Error: no error generated when desired.\n") != SUCCESS)
339                 goto cleanup;
340         }
341     }
342     else
343     {
344         if(tmg_verify_data_valid_error(0x0f, configured_lanes) != SUCCESS)
345         {
346             returncode = FAILURE;
347             (void)lm_error("Sync Error: could not verify error.\n");
348             if(tmg_display_error(0x0f) != SUCCESS) /* all lanes */
349                 goto cleanup;
350         }
351     }
352 }
353
354 /* for each pac in 'configured_lanes'
355 for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++)
356 {
357     if(pac[current_lane].exists == TRUE)
358     {
359         /* Put stop bit in first location of pattern memory */
360

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/mdl_menu_diag.c	DATE 5/23/89	PAGE # 4/35
LINE #	SOURCE TEXT			
361	/*u_loop */(pac(current_lane).lane_offset + BANK_2) = STOP;			
362	pac_set_first_block(configured_lanes, 0);			
363	if(pac_play(TIMOUT) != SUCCESS)			
364	{			
365	returncode = FAILURE;			
366	(void)lm_error("Sync Error: pac_play() returned error.\n");			
367	goto cleanup;			
368	}			
369	if((tmgptr->tmg_intr)			
370	{			
371	returncode = FAILURE;			
372	if(lm_error("Sync Error: did not detect error caused by\n\			
373	\tLane %c with stop bit early.\n", current_lane + 'A') != SUCCESS)			
374	goto cleanup;			
375	}			
376	else			
377	{			
378	if(tmg_verify_data_valid_error(configured_lanes,			
379	Lane_Code(current_lane)) != SUCCESS)			
380	{			
381	returncode = FAILURE;			
382	(void)lm_error("Sync Error: no error with lane %c stopping early.\n",			
383	current_lane + 'A');			
384	if(tmg_display_error(configured_lanes) != SUCCESS)			
385	goto cleanup;			
386	}			
387	if(tmg_clear_error() != SUCCESS)			
388	{			
389	returncode = FAILURE;			
390	(void)lm_error("Cannot reset error.\n");			
391	goto cleanup;			
392	}			
393	if(tmgptr->tmg_intr)			
394	{			
395	returncode = FAILURE;			
396	(void)lm_error("Sync Error: cannot reset error.\n");			
397	goto cleanup;			
398	}			
399	/* Remove stop bit from first location of pattern memory */			
400	/*u_loop */(pac(current_lane).lane_offset + BANK_2) = 0;			
401	}			
402	}			
403	}			
404	}			
405	}			
406	cleanup:			
407	if(tmg_clear_error() != SUCCESS)			
408	{			
409	returncode = FAILURE;			
410	(void)lm_error("Cannot reset error.\n");			
411	}			
412	pac_play_cleanup();			
413	tmgptr->tmg_intr_clear = 0;			
414	tmgptr->tmg_intr_enable = 0;			
415	return(returncode);			
416	}			
417	/*			
418	diag_reset();			
419	/*			
420	OUTPUT: return code = SUCCESS or FAILURE			
421	DESCRIPTION: Resets all lanes in modeler.			
422	*/			
423	diag_reset()			
424	{			
425	return diag_tmg_reset(TRUE);			
426	}			
427	/*			
428	#define BOARDS_IN_LANE (NUMBER_OF_SLOTS + 1)			
429	/*			
430	diag_display()			
431	/*			
432	OUTPUT: return code = SUCCESS or FAILURE			
433	DESCRIPTION: Displays modeler configuration			
434	*/			
435	diag_display()			
436	{			
437	int boards_in_lane;			
438	int failures_in_lane;			
439	int lane_no;			
440	int slot_no;			
441	{			
442	(void)lm_message("LM-1000 HARDWARE CONFIGURATION:\n\n");			
443	(void)diag_display_cpu();			
444	{			
445	if(probe_tmg() != SUCCESS)			
446	{			
447	(void)lm_warning("Insure that CPU/Timing Generator cable is installed.\n");			
448	return(FAILURE);			
449	}			
450	{			
451	(void)diag_display_tmg();			
452	{			
453	/* Fill in exists portion of PAC info structure */			
454	pac_probe_all_pacs();			
455	{			
456	(void)lm_message("\n\t(hit return to continue)\n");			
457	while(lm_get_key() != '\n')			
458	{			
459	{			
460	for(lane_no = 0; lane_no < NUMBER_OF_LANES; ++lane_no)			
461	{			
462	boards_in_lane = 0;			
463	failures_in_lane = 0;			
464	{			
465	(void)lm_message("\tLane %c Information:\n", 'A' + (char)lane_no);			
466	{			
467	if(probe_pac(lane_no) == SUCCESS)			
468	{			
469	boards_in_lane++;			
470	if(diag_display_pac(lane_no) != SUCCESS)			
471	failures_in_lane++;			
472	}			
473	}			
474	{			
475	(void)lm_message("\n");			
476	for(slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++)			
477	{			
478	if(probe_pel(lane_no, slot_no) == SUCCESS)			
479	{			
480	boards_in_lane++;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/mdl\_menu\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:17 pm	5/36

```

LINE #          SOURCE TEXT
481      if(diag_display_pel(lane_no, slot_no) != SUCCESS)
482      {
483      }
484      else {
485      current_lane = lane_no;
486      current_pel = slot_no;
487      pel_dab_seeprom();
488      }
489      }
490      }
491      if(boards_in_lane == 0)
492      (void)lm_message("No boards in lane %c.\n", 'A' + (char)lane_no);
493      else
494      {
495      if((boards_in_lane == BOARDS_IN_LANE) && (failures_in_lane ==
496      BOARDS_IN_LANE))
497      (void)lm_warning("Assure that Timing Generator is fully seated.\n");
498      }
499      (void)lm_message("\n\tHit return to continue.\n");
500      while(lm_get_key() != '\n')
501      ;
502      }
503      return(SUCCESS);
504      }
505      }
506      }
507      }
508      /*
509      * int diag_display_cpu()
510      * INPUT: none
511      * OUTPUT: return code = SUCCESS or FAILURE
512      * DESCRIPTION: Displays CPU board ID from information.
513      * Returns FAILURE if ID from checksum fails.
514      */
515      int
516      diag_display_cpu()
517      {
518      ID_FROM_CPU id_from_table;
519      ID_FROM_CPU *cpu_id_info = &id_from_table;
520      int i;
521      (void)lm_message("CPU board: ");
522      if(diag_get_id_info(CPU_ID_FROM, (char *)cpu_id_info) != SUCCESS)
523      {
524      (void)lm_message("Bad ID FROM\n");
525      return(FAILURE);
526      }
527      diag_display_generic((ID_FROM_GENERIC *)cpu_id_info);
528      /* Display other important info */
529      (void)lm_message("Total memory: %d Mbytes.\n",
530      (cpu_id_info->dram_size == 0x1f) ? 256 : cpu_id_info->dram_size);
531      (void)lm_message("Ethernet address ");
532      for(i = 0; i < 6; i++)
533      (void)lm_message("%02X", cpu_id_info->ethernet[i], (i < 5) ? ' ': '\n');
534      (void)lm_message("Modeler model number: LM-%d.\n",
535      cpu_id_info->model_number);
536      return(SUCCESS);
537      }
538      }
539      }
540      /*
541      * int diag_display_tmg()
542      * INPUT: none
543      * OUTPUT: return code = SUCCESS or FAILURE
544      * DESCRIPTION: Displays Timing Generator ID from information.
545      * Returns FAILURE if ID from checksum fails.
546      */
547      int
548      diag_display_tmg()
549      {
550      ID_FROM_TMG id_from_table;
551      ID_FROM_TMG *tmg_id_info = &id_from_table;
552      (void)lm_message("\nTiming Generator: ");
553      if(diag_get_id_info(TMG_ID_FROM, (char *)tmg_id_info) != SUCCESS)
554      {
555      (void)lm_message("Bad ID FROM\n");
556      return(FAILURE);
557      }
558      diag_display_generic((ID_FROM_GENERIC *)tmg_id_info);
559      return(SUCCESS);
560      }
561      }
562      }
563      }
564      /*
565      * int diag_display_pac(lane_no)
566      * INPUT: lane_no = lane number of Pattern Controller
567      * OUTPUT: return code = SUCCESS or FAILURE
568      * DESCRIPTION: Displays Pattern Controller ID from information.
569      * Returns FAILURE if ID from checksum fails.
570      */
571      int
572      diag_display_pac(lane_no)
573      {
574      int lane_no;
575      ID_FROM_PAC id_from_table;
576      ID_FROM_PAC *pac_id_info = &id_from_table;
577      int pam_no;
578      (void)lm_message("\n Pattern Controller: ");
579      if(diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAC_ID_FROM),
580      (char *)pac_id_info) != SUCCESS)
581      (void)lm_message("Bad ID FROM\n");
582      else
583      diag_display_generic((ID_FROM_GENERIC *)pac_id_info);
584      if(pac_get_num_pams(lane_no) != SUCCESS)
585      {
586      (void)lm_error("Could not display PAM information.\n");
587      return(FAILURE);
588      }
589      for(pam_no = 0; pam_no < pac[lane_no].num_pams; pam_no++)
590      {
591      (void)diag_display_pam(lane_no, pam_no);
592      }
593      return(SUCCESS);
594      }
595      }
596      }
597      }
598      }
599      }
600      /*
601      * int diag_display_pam(lane_no, pam_no)

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/mdl\_menu\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:17 pm	6/37

```

LINE #          SOURCE TEXT
601 *
602 * INPUT: lane_no = lane number of Pattern Controller
603 *
604 * OUTPUT: return code = SUCCESS or FAILURE
605 * DESCRIPTION: Displays Pattern Memory Board ID From information.
606 * Returns FAILURE if ID From checksum fails.
607 */
608 int
609 diag_display_pam(lane_no, pam_no)
610 int lane_no;
611 int pam_no;
612 {
613     ID_FROM_PAM id_prom_table;
614     ID_FROM_PAM *pam_id_info = &id_prom_table;
615     (void)lm_message(" Pattern Memory #td: ", pam_no);
616     if(diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAM_ID_PROM +
617     (pam_no * PAM_ID_SPACE)), (char *)pam_id_info)
618     != SUCCESS)
619     {
620         (void)lm_message("Bad ID FROM\n");
621         return(FAILURE);
622     }
623     (void)lm_message("OK patterns. ", pam_id_info->patterns);
624     diag_display_generic((ID_FROM_GENERIC *)pam_id_info);
625     return(SUCCESS);
626 }
627
628
629
630
631
632 /*      int diag_display_pel()
633 *
634 * INPUT: lane_no = lane number of Pin Electronics Module
635 *        slot_no = slot number of Pin Electronics Module
636 * OUTPUT: return code = SUCCESS or FAILURE
637 * DESCRIPTION: Displays Pin Electronics Module ID From information.
638 * Returns FAILURE if ID From checksum fails.
639 */
640 int
641 diag_display_pel(lane_no, slot_no)
642 int lane_no;
643 int slot_no;
644 {
645     ID_FROM_PEL id_prom_table;
646     ID_FROM_PEL *pel_id_info = &id_prom_table;
647     (void)lm_message(" Pin Electronics Module in slot td: ", slot_no);
648     if(diag_get_id_info((int)(pel_addr(lane_no, slot_no) + PEL_ID_PROM),
649     (char *)pel_id_info) != SUCCESS)
650     {
651         (void)lm_message("Bad ID FROM\n");
652         return(FAILURE);
653     }
654     diag_display_generic((ID_FROM_GENERIC *)pel_id_info);
655     return(SUCCESS);
656 }
657
658
659
660
661
662 int
663 diag_get_id_info(id_prom_address, id_info)
664 int id_prom_address;
665 char *id_info;
666 {
667     u_char *promptr;
668     int byte_count;
669     promptr = (u_char *)(&id_prom_address);
670     if(id_checksum(promptr) != ID_CHECKSUM_GOOD)
671         return(FAILURE);
672     for(byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)
673     {
674         *id_info++ = *promptr;
675         promptr += 4;
676     }
677     return(SUCCESS);
678 }
679
680
681
682 diag_display_generic(id_info)
683 ID_FROM_GENERIC *id_info;
684 {
685     id_stuff *ptr = &(id_info->generic);
686     char eco[32];
687     sprintf(eco, (ptr->eco_level < 32)? "td" : "tc", ptr->eco_level);
688     lm_message("Part no. %03d, Revision %c%c\n",
689     ptr->board_type, ptr->revision, eco);
690 }
691
692

```



Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM diags/modeler_diag.c	DATE	5/23/89	PAGE #
		TIME	4:41:18 pm	1/38

LINE #	SOURCE TEXT
1	/* SCSS_ID: modeler_diag.c rev 3.1, 4/24/89 at 07:48:51 */
2	/*.....*/
3	/*.....*/
4	/*.....*/
5	/*.....*/
6	/*.....*/
7	/*.....*/
8	/*.....*/
9	/*.....*/
10	#include "common.h"
11	#include "lm_diags.h"
12	#include "mod_def.h"
13	#include "modeler_extn.h"
14	#include "tmg.h"
15	#include "tmg_def.h"
16	#include "tmg_extn.h"
17	#include "pac.h"
18	#include "pac_def.h"
19	#include "pac_extn.h"
20	
21	/*
22	* int diag_multi_lane_play()
23	* INPUT = none
24	* OUTPUT = returns SUCCESS or FAILURE
25	* DESCRIPTION: Performs multi-lane play.
26	*/
27	
28	int
29	diag_multi_lane_play()
30	{
31	/* verify no error condition exists now */
32	if(tmgptr->tmg_intr)
33	{
34	(void)lm_error("Multi-lane play : pre-existing error condition.\n");
35	return(FAILURE);
36	}
37	
38	tmgptr->tmg_intr_clear = 1; /* remove interrupt clear */
39	tmgptr->tmg_intr_enable = 1; /* enable error checking */
40	
41	/* setup and play good stuff in all lanes with PACs */
42	for(current_lane = 0; current_lane < NUMBER_OF_LANES; current_lane++)
43	{
44	if(pac(current_lane).exists == TRUE)
45	{
46	(void)pac_fill_pel_ctrl(PATTERNS_IN_128K);
47	}
48	
49	lane_select(configured_lanes);
50	pac_set_first_block(configured_lanes, 0);
51	if(diag_play() != SUCCESS)
52	{
53	(void)lm_error("Multi-lane play : diag_play returned error.\n");
54	pac_play_cleanup();
55	return(FAILURE);
56	}
57	if(tmgptr->tmg_intr)
58	{
59	(void)lm_error("Multi-lane play : presentation caused error.\n");
60	tmg_display_error(configured_lanes);
61	tmg_clear_error();
62	return(FAILURE);
63	}
64	return SUCCESS;
65	}

Copyright 1989 Logic Modeling Systems		HEADER FILE diags/modeler_extn.h	DATE 5/23/89 TIME 4:41:18 pm	PAGE # 1/39
LINE #	HEADER TEXT			
1	/* SOCS_ID: modeler_extn.h rev 3.1, 4/24/89 at 07:48:54 */			
2	/*			
3	.....			
4	modeler_extn.h			
5	.....			
6	External declarations of global variables			
7	used in ls-1000 diagnostics			
8	.....			
9	*/			
10	extern int current_lane; /* current lane being accessed			
11	/* NONE, LANE_A, LANE_B, LANE_C, or LANE_D */			
12	extern int configured_lanes; /* lanes which have configured PACs */			
13	extern long host; /* host == 1 if not running on the modeler */			
14				
15				
16				
17				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/modeler\_glbl.c

DATE 5/23/89

PAGE #

TIME 4:41:18 pm

1/40

```
LINE # SOURCE TEXT
1 /* SCCS_ID: modeler_glbl.c rev 3.1, 4/24/89 at 07:48:57 */
2
3 /*
4  * modeler_glbl.c
5  *
6  * Global variables
7  * used in lw-1000 diagnostics
8  *
9  */
10
11 #include "common.h"
12
13 int current_lane, /* current lane being accessed
14                  * == NONE, LANE_A, LANE_B, LANE_C, or LANE_D */
15
16 int configured_lanes, /* lanes which have configured PACs */
17
18 long Host = 0, /* Host == 1 if not running on the modeler */
19 u_long Host_memory = 0;
```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/modeler\_util.c

DATE	5/23/89	PAGE #
TIME	4:41:18 pm	1/41

```

1  /* SCIS_ID: modeler_util.c rev 3.1, 4/24/89 at 07:49:00 */
2  .....
3  *
4  *   modeler_util.c
5  *
6  *   Utility routines
7  *   used in diagnostics
8  *
9  *   .....
10 #include "common.h"
11 #include "lm_diags.h"
12 #include "mod_def.h"
13 #include "modeler_extn.h"
14 #include "vrtx.h"
15 #include "tmg.h"
16 #include "tmg_def.h"
17 #include "tmg_extn.h"
18 #include "pac.h"
19 #include "pac_def.h"
20 #include "pac_extn.h"
21 #include "id.h"
22
23 int lm_message(), lm_warning(), lm_error();
24
25 /*
26  *   int diag_play()
27  *
28  *   INPUT: none
29  *   OUTPUT: returns SUCCESS or FAILURE
30  *   DESCRIPTION: Sees if there are any backplane errors and attempts
31  *   to clear any that exist. Returns FAILURE if backplane error still
32  *   exists.
33  */
34
35 int
36 diag_clear_errors()
37 {
38     int lane;
39     int pel_no;
40     int returncode = SUCCESS;
41
42     /* See if there are any backplane errors */
43     if(Get_bp_error() == 0)
44         return(SUCCESS);
45     /* There are errors on the backplane */
46     for(lane = 0; lane < NUMBER_OF_LANES; ++lane)
47     {
48         if((Lane_code(lane) & Get_bp_error()) != 0)
49         {
50             /* See what's driving the error line and attempt to clear the error */
51             if(probe_pac(lane) == SUCCESS)
52             {
53                 if(pac_check_errors(lane, lm_message) != SUCCESS)
54                     Clear_pac_errors(lane);
55             }
56             pel_no = 0;
57             while(((Lane_code(lane) & Get_bp_error()) != 0) &&
58                 (pel_no < NUMBER_OF_PELS))
59             {
60                 if(probe_pel(lane, pel_no) == SUCCESS)
61                 {
62                     if(pel_check_errors(lane, pel_no, lm_message) != SUCCESS)
63                         pel_disable_bp_error(lane, pel_no);
64                     ++pel_no;
65                 }
66             }
67             if((Lane_code(lane) & Get_bp_error()) != 0)
68             {
69                 returncode = FAILURE;
70                 (void)lm_error("Unable to remove backplane error from lane %c.\n",
71                     (char)lane + 'A');
72             }
73         }
74     }
75     return(returncode);
76 }
77
78 report_bp_error()
79 {
80     int lane;
81     int pel_no;
82     int returncode;
83
84     /* See if there are any backplane errors */
85     if((returncode = Get_bp_error()) == 0)
86         return(returncode);
87     /* There are errors on the backplane */
88     for(lane = 0; lane < NUMBER_OF_LANES; ++lane)
89     {
90         if((Lane_code(lane) & Get_bp_error()) != 0)
91         {
92             (void)lm_error("There is a backplane error in lane %c.\n",
93                 (char)lane + 'A');
94             /* See what's driving the error line */
95             if(probe_pac(lane) == SUCCESS)
96             {
97                 pac_check_errors(lane, lm_error);
98             }
99             for(pel_no = 0; pel_no < NUMBER_OF_PELS; ++pel_no)
100             {
101                 if(probe_pel(lane, pel_no) == SUCCESS)
102                     pel_check_errors(lane, pel_no, lm_error);
103             }
104         }
105     }
106     return(returncode);
107 }
108
109 /*
110  *   int diag_play()
111  *
112  *   INPUT: none
113  *   OUTPUT: returns SUCCESS or FAILURE
114  *   DESCRIPTION: Sets up PAC clock speed reg, computes timeout,
115  *   and performs a pattern play.
116  */
117
118 int
119 diag_play()
120 {
121     int timeout;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/modeler_util.c	DATE 5/23/89	PAGE # 2/42
LINE #	SOURCE TEXT			
121	if((timeout = pac_pre_play()) == 0)			
122	{			
123	(void)lm_error("Unable to prepare for pattern play.\n");			
124	return(FAILURE);			
125	}			
126	if(pac_play(timeout) != SUCCESS)			
127	{			
128	(void)lm_error("Pattern play fails.\n");			
129	return(FAILURE);			
130	}			
131	return(SUCCESS);			
132	}			
133				
134				
135	/*     void diag_get_long(long_value, &prompt, low, high)			
136	/*			
137	/*     INPUT:   long_value = address of long value			
138	/*            &prompt = address of input prompt			
139	/*            low = lower bound of input (input >= low)			
140	/*            high = upper bound of input (input <= high)			
141	/*     OUTPUT: none			
142	/*     DESCRIPTION: Gets long value from keyboard.			
143	/*     Checks bounds on input.			
144	*/			
145	void			
146	diag_get_long(long_value, prompt, low, high)			
147	long *long_value;			
148	char *prompt;			
149	long low;			
150	long high;			
151	{			
152	char reply(DIAG_MAX_INPUT);			
153	char buffer[1024];			
154	char *extra;			
155	register long answer_val;			
156				
157	/* Get input from keyboard */			
158	sprintf(buffer, "Enter %s (%d): ", prompt, *long_value);			
159	do			
160	{			
161	lm_get_input(buffer, reply, DIAG_MAX_INPUT);			
162	if (reply[0] == '\0')			
163	return;			
164	answer_val = strtol(reply, &extra, 0);			
165	if(("extra == NULL) && (answer_val >= low) && (answer_val <= high))			
166	{			
167	*long_value = answer_val;			
168	return;			
169	}			
170	(void)lm_message("%d <= value <= %d\n", low, high);			
171	} while(1);			
172				
173				
174	/*     void diag_get_ulong(long_value, &prompt, low, high)			
175	/*			
176	/*     INPUT:   ulong_value = address of unsigned long value			
177	/*            &prompt = address of input prompt			
178	/*            low = lower bound of input (input >= low)			
179	/*            high = upper bound of input (input <= high)			
180	/*     OUTPUT: none			
181	/*     DESCRIPTION: Gets unsigned long value from keyboard.			
182	/*     Checks bounds on input.			
183	*/			
184	void			
185	diag_get_ulong(ulong_value, prompt, low, high)			
186	ulong *ulong_value;			
187	char *prompt;			
188	ulong low;			
189	ulong high;			
190	{			
191	char reply(DIAG_MAX_INPUT);			
192	char buffer[1024];			
193	char *extra;			
194	register ulong answer_val;			
195				
196	/* Get input from keyboard */			
197	sprintf(buffer, "Enter %s (%u): ", prompt, *ulong_value);			
198	do			
199	{			
200	lm_get_input(buffer, reply, DIAG_MAX_INPUT);			
201	if (reply[0] == '\0')			
202	return;			
203	answer_val = (ulong)strtoul(reply, &extra, 0);			
204	if(("extra == NULL) && (answer_val >= low) && (answer_val <= high))			
205	{			
206	*ulong_value = answer_val;			
207	return;			
208	}			
209	(void)lm_message("%u <= value <= %u\n", low, high);			
210	} while(1);			
211				
212				
213	/*     void diag_get_uxax(ulong_value, &prompt, low, high)			
214	/*			
215	/*     INPUT:   ulong_value = address of unsigned long hex value			
216	/*            &prompt = address of input prompt			
217	/*            low = lower bound of input (input >= low)			
218	/*            high = upper bound of input (input <= high)			
219	/*     OUTPUT: none			
220	/*     DESCRIPTION: Gets unsigned long hex value from keyboard.			
221	/*     Checks bounds on input.			
222	*/			
223	void			
224	diag_get_uxax(ulong_value, prompt, low, high)			
225	ulong *ulong_value;			
226	char *prompt;			
227	ulong low;			
228	ulong high;			
229	{			
230	char reply(DIAG_MAX_INPUT);			
231	char buffer[1024];			
232	char *extra;			
233	register ulong answer_val;			
234				
235	/* Place a leading 0x in front of the input */			
236	reply_ptr++ = '0';			
237	reply_ptr++ = 'x';			
238	/* Get input from keyboard */			
239	sprintf(buffer, "Enter %s (%X): ", prompt, *ulong_value);			
240				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/modeler\_util.c

DATE 5/23/89  
TIME 4:41:18 pm

PAGE #  
3/43

```

LINE #          SOURCE TEXT
241 do
242 {
243     in_get_input(buffer, reply_ptr, DIAG_MAX_INPUT);
244     if (*reply_ptr == '\0')
245         return;
246     answer_val = (u_long)strtol((reply_ptr + 2), &extra, 0);
247     if ((*extra == NULL) && (answer_val >= low) && (answer_val <= high))
248     {
249         *ulong_value = answer_val;
250         return;
251     }
252     (void)in_message("XI <= value <= XI\n", low, high);
253     while(1);
254 }
255
256 /*
257  *   int id_checksum(address)
258  *   INPUT:  address - pointer to first location of ID FROM
259  *           (Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart)
260  *   OUTPUT: returns ID FROM checksum (8 bits)
261  *   DESCRIPTION: Computes ID FROM checksum.
262  *   Algorithm:
263  *   - initialise checksum to an arbitrary (but well-known) value
264  *   - for each byte (including checksum)
265  *   -   circular left shift left checksum
266  *   -   add data byte
267  *   -   mask checksum to 8 bits
268  */
269 int
270 id_checksum(address)
271 u_char *address;
272 {
273     register int checksum;
274     register u_long byte_count;
275
276     checksum = ID_CHECKSUM_INIT;
277
278     for (byte_count = 0; byte_count < ID_NUM_BYTES; byte_count++)
279     {
280         checksum = (checksum << 1) + ((checksum & 0x80) >> 7);
281         checksum += *(address + 4 * byte_count);
282     }
283     checksum = 0xFF;
284     return(checksum);
285 }
286

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:19 pm

1/44

```

1  /* SCCS ID: pac_diag.c rev 3.1, 4/24/89 at 07:49:03 */
2
3  .....
4  *   pac_diag.c
5  *
6  *   Diagnostic routine called by PAC menu functions
7  *   used in PAC diagnostics
8  *
9  .....
10 #include "common.h"
11 #include "mod_def.h"
12 #include "modeler_extn.h"
13 #include "tmg.h"
14 #include "tmg_def.h"
15 #include "tmg_extn.h"
16 #include "pac.h"
17 #include "pac_def.h"
18 #include "pac_extn.h"
19 #include "id.h"
20
21
22 /*
23  *   : int pac_parity_test(address)
24  *
25  *   INPUT: address = pattern memory address (long word)
26  *   OUTPUT: returns code = SUCCESS or FAILURE
27  *   DESCRIPTION: Performs extensive parity test on long word
28  *   specified by 'address'. Returns SUCCESS if test passes,
29  *   FAILURE if test fails.
30  */
31 int
32 pac_parity_test(address)
33     u_long address;
34 {
35     int returncode = SUCCESS;
36
37     /* Check for existing parity errors and clear them if nec */
38     if((pacptr[current_lane] -> high_word_parity_error == TRUE) ||
39        (pacptr[current_lane] -> low_word_parity_error == TRUE)) {
40         clear_pac_errors(current_lane);
41     }
42     /* Perform parity tests, writing and reading with same parity */
43     pacptr[current_lane] -> high_word_parity = SET_EVEN_PARITY;
44     if(good_parity_check(address) != SUCCESS) /* High word, even parity */
45     {
46         returncode = FAILURE;
47         if(lm_error("PAC parity test (high word, even parity).\n") != SUCCESS)
48             goto cleanup;
49     }
50     pacptr[current_lane] -> high_word_parity = SET_ODD_PARITY;
51     if(good_parity_check(address) != SUCCESS) /* High word, odd parity */
52     {
53         returncode = FAILURE;
54         if(lm_error("PAC parity test (high word, odd parity).\n") != SUCCESS)
55             goto cleanup;
56     }
57     pacptr[current_lane] -> low_word_parity = SET_EVEN_PARITY;
58     if(good_parity_check(address + 2) != SUCCESS) /* Low word, even parity */
59     {
60         returncode = FAILURE;
61         if(lm_error("PAC parity test (low word, even parity).\n") != SUCCESS)
62             goto cleanup;
63     }
64     pacptr[current_lane] -> low_word_parity = SET_ODD_PARITY;
65     if(good_parity_check(address + 2) != SUCCESS) /* Low word, odd parity */
66     {
67         returncode = FAILURE;
68         if(lm_error("PAC parity test (low word, odd parity).\n") != SUCCESS)
69             goto cleanup;
70     }
71     /* Perform parity tests, writing with odd parity */
72     /* and reading with even parity */
73     if(bad_parity_check(address) != SUCCESS) /* High word parity */
74     {
75         returncode = FAILURE;
76         if(lm_error("PAC parity test (high word, odd/even).\n") != SUCCESS)
77             goto cleanup;
78     }
79     if(bad_parity_check(address + 2) != SUCCESS) /* Low word parity */
80     {
81         returncode = FAILURE;
82         if(lm_error("PAC parity test (low word, odd/even).\n") != SUCCESS)
83             goto cleanup;
84     }
85     cleanup:
86     /* Restore parity circuits to ODD parity */
87     pac_set_parity(current_lane, SET_ODD_PARITY);
88     return(returncode);
89 }
90
91 /*
92  *   void pac_build_fast_branch()
93  *
94  *   INPUT: none
95  *   OUTPUT: none
96  *   DESCRIPTION: Places branch always instructions in
97  *   second location of each block (except block near
98  *   center of pattern memory). The link table is set up
99  *   to branch from the outermost blocks toward the center
100  *   of the memory, starting from the last block.
101  */
102 void
103 pac_build_fast_branch()
104 {
105     u_long *memptr;
106     u_long block;
107     u_long i;
108
109     memptr = (u_long *) (pac[current_lane].lane_offset + BANK_2 + 4);
110     /* Put branch instructions in second location of each block (minimum) */
111     for(block = 0; block < pac[current_lane].num_blocks; ++block)
112     {
113         *memptr |= BRANCH_ALWAYS;
114         memptr += BLOCK_SIZE;
115     }
116     /* Remove branch command in "middle" of pattern memory and */
117     /* place stop command near branch location */
118     memptr = (u_long *) (pac[current_lane].lane_offset + BANK_2 +
119 4 * ((pac[current_lane].num_blocks/2 - 1) * BLOCK_SIZE + 1));
120     *memptr |= NOP_MASK;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:19 pm

2/45

```

121  *sample_ptr += BRANCH_LATENCY - STOP_LATENCY;
122  *sample_ptr = STOP;
123
124  /* Fill up link table to branch toward the center of memory */
125  for(block = 0; block < pac(current_lane).sum_blocks - 2; block++)
126  {
127      block = ((pac(current_lane).sum_blocks > 1) - 1) / 2;
128      *sample_ptr++ = block;
129  }
130  /* Set branch address register to start with last block */
131  pacptr(current_lane) = pac(current_lane).sum_blocks - 1;
132
133  /* Set branch address register to start with last block */
134  pacptr(current_lane) = pac(current_lane).sum_blocks - 1;
135
136  /*
137   * int pec_sweep_test(patterns);
138   * INPUT: patterns = number of patterns to be played
139   * OUTPUT: return code = SUCCESS or FAILURE
140   * DESCRIPTION: Plays patterns at various frequencies,
141   * measuring time of play and comparing with expected
142   * time.
143   */
144
145  int pec_sweep_test(patterns)
146  {
147      int actual_time;
148      int period_step;
149      int period;
150      int i;
151      int returncode = SUCCESS;
152      void setup_good_sample();
153
154      /* Sweep from lowest frequency to 1 MHz (10 steps) */
155      pacptr(current_lane) = clock_speed = BELOW_1MHZ;
156      period_step = (PAC_MAX_PERIOD - 1000) / 9;
157      for(period = 1000, i = 0; i < 10; period += period_step, i++)
158      {
159          (void)lm_message(" ");
160          if(pec_set_pattern_clock(period) != SUCCESS)
161          {
162              (void)lm_error("Sweep test could not set pattern clock.\n");
163              return(FAILURE);
164          }
165          setup_good_sample(period = 1000); /* needs period in ps */
166          sample_width = pec_compute_sample_width(period);
167          if(pec_timed_play(period, patterns, actual_time) != SUCCESS)
168          {
169              returncode = FAILURE;
170              if(lm_error("Play timing failed during sweep test. Clock period = %d.\n",
171                  period) != SUCCESS)
172                  return(FAILURE);
173          }
174      }
175
176      /* Sweep from 1 MHz to highest frequency (10 steps) */
177      pacptr(current_lane) = clock_speed = ABOVE_1MHZ;
178      period_step = (1000 - PAC_MIN_PERIOD) / 9;
179      for(period = 1000, i = 0; i < 10; period -= period_step, i++)
180      {
181          (void)lm_message(" ");
182          if(pec_set_pattern_clock(period) != SUCCESS)
183          {
184              (void)lm_error("Sweep test could not set pattern clock.\n");
185              return(FAILURE);
186          }
187          setup_good_sample(period = 1000); /* needs period in ps */
188          sample_width = pec_compute_sample_width(period);
189          if(pec_timed_play(period, patterns, actual_time) != SUCCESS)
190          {
191              returncode = FAILURE;
192              if(lm_error("Play timing failed during sweep test. Clock period = %d.\n",
193                  period) != SUCCESS)
194                  return(FAILURE);
195          }
196      }
197
198      (void)lm_message("done.\n");
199      return(returncode);
200
201  /*
202   * int good_parity_check(address)
203   * INPUT: address = word address of pattern memory
204   * OUTPUT: return code = SUCCESS or FAILURE
205   * DESCRIPTION: Checks parity circuitry on PAC and PAM
206   * by writing and reading the location specified by 'address'.
207   * The location is written and read 32 times with a pattern of
208   * shifting ones and zeros.
209   */
210
211  int good_parity_check(address)
212  {
213      register u_long address;
214      u_long i;
215      int j;
216      u_short temp;
217      int returncode = SUCCESS;
218
219      for(i=0; i<32; i++)
220      {
221          Write_word(address, i);
222          temp = Read_word(address);
223          if((pacptr(current_lane) -> high_word_parity_error == TRUE) ||
224             (pacptr(current_lane) -> low_word_parity_error == TRUE))
225          {
226              returncode = FAILURE;
227              Clr_pac_errors(current_lane);
228              if(lm_error("PAC 'good' Parity test. Address= %08x, DATA= %04x.\n",
229                  address, i) != SUCCESS)
230                  return(FAILURE);
231          }
232      }
233
234  #ifdef LIST
235      if (temp); /* shut list up */
236  #endif
237      return(returncode);
238

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:19 pm 3/46

```

LINE # SOURCE TEXT
241 )
242
243 /* int bad_parity_check(address)
244 *
245 * INPUT: address = word address of pattern memory
246 * OUTPUT: return code = SUCCESS or FAILURE
247 * DESCRIPTION: Checks parity circuitry on PAC and PAM
248 * by writing and reading the location specified by 'address'.
249 * The location is written and read 32 times with a pattern of
250 * shifting ones and zeros. The location is written with the
251 * parity set to 'odd' and then read with the parity set to
252 * 'even'. This should create a parity error, latching the
253 * address. The latched address is compared with the actual
254 * address to see if this circuitry is working.
255 */
256
257 int
258 bad_parity_check(address)
259 register u_long address;
260 {
261     u_long i;
262     int j;
263     u_long read_addr;
264     u_long expect_addr;
265     u_short temp;
266     int returncode = SUCCESS;
267
268     switch(address & 0x02) /* High or low word? */
269     {
270     case 0: /* High word test */
271         for(i=0, j=0; j < 32; i += (j++ & 0x02))
272         {
273             pacptr[current_lane] = high_word_parity = SET_ODD_PARITY;
274             Write_word(address,i);
275             pacptr[current_lane] = high_word_parity = SET_EVEN_PARITY;
276             temp = Read_word(address);
277             if((pacptr[current_lane] -> high_word_parity_error == TRUE) &&
278                 (pacptr[current_lane] -> low_word_parity_error == FALSE))
279             {
280                 if((read_addr = pacptr[current_lane] -> parity_error_address) !=
281                     (expect_addr = (address >> 2) & 0x3FFFFFF))
282                 {
283                     returncode = FAILURE;
284                     if(!m_error("PAC parity address test. Expected = %07x, \
285                         Actual = %07x.\n", expect_addr, read_addr) != SUCCESS)
286                         return(FAILURE);
287                 }
288             }
289             else
290             {
291                 returncode = FAILURE;
292                 if(!m_error("PAC 'bad' parity test. Address = %08x, Data = %04x.\n",
293                     address, i) != SUCCESS)
294                     return(FAILURE);
295             }
296             Clear_pac_errors(current_lane);
297         }
298         break;
299     case 2: /* Low word test */
300         for(i=0, j=0; j < 32; i += (j++ & 0x02))
301         {
302             pacptr[current_lane] = low_word_parity = SET_ODD_PARITY;
303             Write_word(address,i);
304             pacptr[current_lane] = low_word_parity = SET_EVEN_PARITY;
305             temp = Read_word(address);
306             if((pacptr[current_lane] -> high_word_parity_error == FALSE) &&
307                 (pacptr[current_lane] -> low_word_parity_error == TRUE))
308             {
309                 if((read_addr = pacptr[current_lane] -> parity_error_address) !=
310                     (expect_addr = (address >> 2) & 0x3FFFFFF))
311                 {
312                     returncode = FAILURE;
313                     if(!m_error("PAC parity address test. Expected = %07x, \
314                         Actual = %07x.\n", expect_addr, read_addr) != SUCCESS)
315                         return(FAILURE);
316                 }
317             }
318             else
319             {
320                 returncode = FAILURE;
321                 if(!m_error("PAC 'bad' parity test. Address = %08x, Data = %04x.\n",
322                     address, i) != SUCCESS)
323                     return(FAILURE);
324             }
325             Clear_pac_errors(current_lane);
326         }
327         break;
328     default: /* Bad address */
329         return(FAILURE);
330     }
331
332     if(!m_error("PAC parity test. Address = %08x, Data = %04x.\n",
333         address, temp) != SUCCESS)
334         return(FAILURE);
335     return(returncode);
336 }
337
338 /* int pam_idprom_test(lane_no)
339 *
340 * INPUT: lane_no = lane number to check PAM ID PROMS in
341 * OUTPUT: return code = SUCCESS or FAILURE
342 * DESCRIPTION: Performs checksum tests on PAM ID PROMs.
343 * The function returns SUCCESS if all of the checksums
344 * are correct, and returns FAILURE if any of the checksum
345 * tests fail.
346 */
347 int
348 pam_idprom_test(lane_no)
349 int lane_no;
350 {
351     int temp;
352     int pamno;
353     u_long base_address;
354     int returncode = SUCCESS;
355
356     base_address = pac[lane_no].lane_offset + PAM_ID;
357     for(pamno = 0; pamno < pac[lane_no].num_pams; ++pamno)
358     {
359         if((temp = id_checksum((u_char *) (base_address + pamno * PAM_ID_SPACE + 3)))
360             != ID_CHECKSUM_GOOD)

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_diag.c	DATE 5/23/89	PAGE # 4/47
LINE #	SOURCE TEXT			
361	{			
362	returncode = FAILURE;			
363	if(!m_error("PAM ad ID FROM checksum. Expected = %02x. Actual = %02x.\n",			
364	pamno, ID_CHECKSUM_GOOD, temp) != SUCCESS)			
365	return(FAILURE);			
366	}			
367	}			
368	return(returncode);			
369	}			
370	}			
371	/*			
372	int pac_comp_pel_ctrl(patterns_no, bit_no)			
373	/*			
374	INPUT: patterns_no = patterns number, 0's counting			
375	bit_no = PEL control bit number (0, 1, or 2)			
376	OUTPUT: return code = SUCCESS or FAILURE			
377	DESCRIPTION: Complements the PEL control bit specified by			
378	bit_no in the patterns specified by patterns_no. The function			
379	checks if the patterns number is within patterns memory.			
380	*/			
381	int			
382	pac_comp_pel_ctrl(patterns_no, bit_no)			
383	u_long patterns_no;			
384	int bit_no;			
385	{			
386	u_long address;			
387	if((patterns_no < 0)    (patterns_no > (pac(current_lane).num_patterns - 1)))			
388	{			
389	(void)m_error("Pattern outside of pattern memory.\n");			
390	return(FAILURE);			
391	}			
392	address = pac(current_lane).lane_offset + BANK_2 + (4 * patterns_no);			
393	*(u_long *)address = 1 << (4 + bit_no);			
394	return(SUCCESS);			
395	}			
396	/*			
397	int pac_fill_pel_ctrl(patterns)			
398	/*			
399	INPUT: patterns = number of patterns to fill			
400	OUTPUT: return code = SUCCESS or FAILURE			
401	DESCRIPTION: Fills the PEL control bits with a counting			
402	pattern for the number of patterns specified. Patterns			
403	control is also written.			
404	*/			
405	int			
406	pac_fill_pel_ctrl(patterns)			
407	int patterns;			
408	{			
409	register u_long *sumptr;			
410	register int pat_no;			
411	if((patterns < 4)    (patterns > pac(current_lane).num_patterns))			
412	{			
413	(void)m_error("Number of patterns invalid.\n");			
414	return(FAILURE);			
415	}			
416	sumptr = (u_long *) (pac(current_lane).lane_offset + BANK_2);			
417	for(pat_no = 0; pat_no < patterns; pat_no++)			
418	{			
419	*sumptr++ = (pat_no & 0x7) << 4;			
420	}			
421	if(build_patterns_control(0, patterns, STOP_MODE) != SUCCESS)			
422	{			
423	(void)m_error("Could not build pattern control.\n");			
424	return(FAILURE);			
425	}			
426	return(SUCCESS);			
427	}			
428	/*			
429	int pac_predict_offsets(patterns, error_latency, branch_offset,			
430	block_offset)			
431	/*			
432	INPUT: patterns = patterns number with parity error			
433	error_latency = number of patterns played after parity error			
434	branch_offset = address of predicted branch offset			
435	block_offset = address of predicted block offset			
436	OUTPUT: return code = SUCCESS or FAILURE			
437	DESCRIPTION: Gives the specified patterns number, this function			
438	determines what the contents of the branch address and block offset			
439	registers would be after a pattern play. The function returns			
440	FAILURE if the pattern would not be played because of a STOP			
441	instruction in the block. Note: there must not be a parity error			
442	in patterns memory when this function is called. Nor should there			
443	be a parity error condition.			
444	*/			
445	int			
446	pac_predict_offsets(patterns, error_latency, branch_offset, block_offset)			
447	int patterns;			
448	int error_latency;			
449	int branch_offset;			
450	int block_offset;			
451	{			
452	u_long *sumptr;			
453	int block[2];			
454	register int i;			
455	int branch_no;			
456	int stop_no;			
457	int played;			
458	int patterns_offset;			
459	/* Figure out which block number patterns is in */			
460	block[0] = patterns / BLOCK_SIZE;			
461	/* Figure out offset into block */			
462	patterns_offset = patterns % BLOCK_SIZE;			
463	/* Assign "most probable" values to returned arguments (may be modified			
464	later in the function) */			
465	branch_offset = block[0];			
466	block_offset = (patterns_offset + error_latency) % BLOCK_SIZE;			
467	/* Search for branch and stop instructions */			
468	/* (assumes one and only one branch and/or stop per block) */			
469	sumptr = (u_long *) (pac(current_lane).lane_offset + BANK_2 +			
470	block[0] * BLOCK_SIZE * 4);			
471	branch_no = -1;			
472	stop_no = -1;			
473	for(i = 0; i < BLOCK_SIZE * 4; i++)			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:19 pm 5/48

```

LINE #      SOURCE TEXT
481      {
482      if ((*sampleptr & BRANCH_ALWAYS) != 0)
483      {
484      branch_no = 1;
485      break;
486      }
487      if ((*sampleptr++ & STOP) != 0)
488      {
489      stop_no = 1;
490      break;
491      }
492      }
493      if (branch_no == -1) /* no branch found => only a stop command */
494      {
495      /* See if pattern gets played */
496      if ((stop_no + STOP_LATENCY) > pattern_offset) /* pattern is played */
497      return(SUCCESS);
498      else /* pattern is not played */
499      return(FAILURE);
500      }
501      /* At this point there is at least a branch command */
502      /* See if pattern with error is played before/during branch */
503      if ((branch_no + BRANCH_LATENCY) > pattern_offset) /* pattern is played */
504      {
505      /* See if branch command is reached before halt */
506      if ((pattern_offset + error_latency - 2) > branch_no) /* reached */
507      {
508      /* At this point, we can compute the block offset. We still do
509      not know the correct branch offset. The block offset is
510      either correct, or if it occurs near the end of a branch, must
511      be modified. */
512      if ((played + pattern_offset - (branch_no + BRANCH_LATENCY -
513      error_latency - 4)) > 0) /* need to modify block_offset */
514      *block_offset = (played + 0x1c) & BLOCK_SIZE;
515      /* Now we must determine the correct block number */
516      /* Look up next block */
517      block[1] = Read_long(pac[current_lane].lane_offset + LINK_OFFSET +
518      block[0] * 4) & 0x7fff;
519      /* See if block 2 is entered before halt */
520      /* First, see how many patterns are played in block 1, if any */
521      if ((played - (pattern_offset + error_latency) -
522      (branch_no + BRANCH_LATENCY + 2)) > 0) /* patterns played in block 1 */
523      {
524      /* See if there is a branch command within the first "played"
525      patterns in block 1, to see if we branch into block 2 */
526      sampleptr = (u_long *) (pac[current_lane].lane_offset + BANK_2 +
527      block[1] * BLOCK_SIZE * 4);
528      for (i = 0; i < played; i++)
529      {
530      if ((*sampleptr++ & BRANCH_ALWAYS) != 0)
531      break;
532      }
533      if (i == played) /* Did not get to block 2 */
534      {
535      *branch_offset = block[1];
536      return(SUCCESS);
537      }
538      else /* Got to block 2 */
539      {
540      *branch_offset = Read_long(pac[current_lane].lane_offset +
541      LINK_OFFSET + block[1] * 4) & 0x7fff;
542      return(SUCCESS);
543      }
544      }
545      else /* no patterns played in block 1 */
546      {
547      *branch_offset = block[1];
548      return(SUCCESS);
549      }
550      }
551      else /* branch is not reached */
552      return(SUCCESS);
553      }
554      else /* pattern is not played */
555      return(FAILURE);
556      }
557      }
558      }
559      }
560      }
561      }
562      }
563      }
564      }
565      }
566      }
567      }
568      }
569      }
570      }
571      }
572      }
573      }
574      }
575      }
576      }
577      }
578      }
579      }
580      }
581      }
582      }
583      }
584      }
585      }
586      }
587      }
588      }
589      }
590      }
591      }
592      }
593      }
594      }
595      }
596      }
597      }
598      }
599      }
600      }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_diag.c		DATE 5/23/89	PAGE # 6/49
TIME 4:41:19 pm					
SOURCE TEXT					
LINE #					
601	{				
602	int ramp;				
603	/* find the fastest safe ramp to use for Edge7 */				
604	for(ramp = 0; ramp < 3; ramp++)				
605	if(calib.EdgeMaxDelay[ramp] > period)				
606	break;				
607	/* set up some stuff on the TMC */				
608	tmcptr->edge_delay_range = ramp;				
609	tmcptr->sample_delay_range = 0; /* choose fastest ramp */				
610	tmcptr->sample_mode = EDGE7SAMPLETRIGGERMODE;				
611	tmcptr->sample_trigger_threshold = calib.Edge7MinThresh[ramp];				
612	tmcptr->sample_delay = calib.SampleMinThresh(0);				
613	}				

Copyright 1989 Logic Modeling Systems		HEADER FILE diags/pac_extn.h	DATE 5/23/89	PAGE # 1/50
			TIME 4:41:19 pm	
LINE #	HEADER TEXT			
1	/* SCCS_ID: pac_extn.h rev 3.1, 4/24/89 at 07:49:07 */			
2	/*			
3	*****			
4	pac_extn.h			
5	/*			
6	External declarations of global variables			
7	used in PAC diagnostics			
8	*****			
9	/*			
10	External variables */			
11	extern PAC *pacptr(NUMBER_OF_LANES); /* PAC board (one for each lane) */			
12	extern PAC_INFO pac(NUMBER_OF_LANES); /* PAC information table */			
13				

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_glbl.c

DATE

5/23/89

PAGE #

TIME

4:41:19 pm

1/51

LINE #	SOURCE TEXT
1	/* SCOS_ID: pac_glbl.c rev 3.1, 4/24/89 at 07:49:10 */
2	/*
3	.....
4	pac_glbl.c
5	.....
6	Global variables
7	used in PAC diagnostics
8	.....
9	*/
10	#include "common.h"
11	#include "mod_def.h"
12	#include "pac.h"
13	#include "pac_def.h"
14	
15	PAC "pacptr(NUMBER_OF_LANES), /* PAC boards (one for each lane) */
16	PAC_INFO pac(NUMBER_OF_LANES), /* PAC information table */

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_memtest.c

DATE 5/23/89  
TIME 4:41:19 pm

PAGE #  
1/52

```

1  /* SCCS ID: pac_memtest.c rev 3.1, 4/24/89 at 07:49:15 */
2  /*
3  * .....
4  *   pac_memtest.c
5  * .....
6  *   Memory test routines
7  *   used in PAC diagnostics
8  * .....
9  */
10 #include "common.h"
11 #include "mod_def.h"
12 #include "modeler_exta.h"
13 #include "pac.h"
14 #include "pac_def.h"
15 #include "pac_exta.h"
16
17 typedef struct
18 {
19     u_long bad_data_bits;
20     u_long vram1thru24;
21     u_long vram27thru50;
22     int bad_parity_errors;
23 } MEM_ERRORS;
24
25
26
27
28 /*
29 * void pac_init_mem_err(error_ptr)
30 *
31 * INPUT: error_ptr = pointer to memory error structure
32 * OUTPUT: none
33 * DESCRIPTION: Initializes pettara memory test error structure.
34 */
35 void
36 pac_init_mem_err(error_ptr)
37 MEM_ERRORS *error_ptr;
38 {
39     error_ptr->bad_data_bits = 0L;
40     error_ptr->vram1thru24 = 0L;
41     error_ptr->vram27thru50 = 0L;
42     error_ptr->bad_parity_errors = FALSE;
43 }
44
45
46 /*
47 * void pac_display_mem_err(error_ptr)
48 *
49 * INPUT: error_ptr = pointer to memory error structure
50 * OUTPUT: none
51 * DESCRIPTION: Displays pettara memory test error structure.
52 */
53 void
54 pac_display_mem_err(error_ptr)
55 MEM_ERRORS *error_ptr;
56 {
57     register int i;
58     register u_long bad_ics;
59     register int word;
60     register int sum_printed = 0;
61
62     (void)lm_message("\nPettara Memory Failure Summary:\n\n");
63     if(error_ptr->bad_data_bits != 0)
64     {
65         (void)lm_message("Bits which did not compare: %08X\n",
66             error_ptr->bad_data_bits);
67         for(word = 0; word < 2; ++word) /* word=0 => low word */
68         {
69             bad_ics = (word ? error_ptr->vram1thru24 : error_ptr->vram27thru50);
70             if(bad_ics != 0)
71             {
72                 (void)lm_message("\n%08X word memory ICs:\n", (word ? "High" : "Low"));
73                 sum_printed = 0;
74                 for(i = 1; i < 25; ++i)
75                 {
76                     if(((1 << i) & bad_ics) != 0)
77                     {
78                         (void)lm_message("  %04d", (word ? i : i + 24));
79                         ++sum_printed;
80                     }
81                     if(sum_printed > 12)
82                     {
83                         (void)lm_message("\n");
84                         sum_printed = 0;
85                     }
86                 }
87             }
88         }
89         if(sum_printed != 0)
90             (void)lm_message("\n");
91         if(error_ptr->bad_parity_errors == TRUE)
92             (void)lm_message("Parity memory is suspect.\n");
93     }
94 }
95
96 /*
97 * int data_bus_test(address, error_ptr)
98 *
99 * INPUT: address = memory address
100 * error_ptr = pointer to memory error structure
101 * OUTPUT: returns SUCCESS or FAILURE
102 * DESCRIPTION: Performs data bus test on a 32-bit memory location.
103 */
104 int
105 data_bus_test(address, error_ptr)
106 u_long address;
107 MEM_ERRORS *error_ptr;
108 {
109     register u_long *memptr = (u_long *)address;
110     register int i;
111     register u_long expected;
112     register u_long actual;
113     register int returncode = SUCCESS;
114     int invert_data;
115     for(invert_data = 0; invert_data < 2; invert_data++)
116     {
117         for(i = 1; i <= (1 << 31); i <= 1)
118         {
119             expected = (invert_data ? ~i : i);
120             *memptr = expected;
121         }
122     }
123 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_memtest.c

DATE	5/23/89	PAGE #
TIME	4:41:19 pm	2/53

```

121  if((actual - *memptr) != expected)
122  {
123      returncode = FAILURE;
124      if(pac_mem_error((u_long)(memptr - 1), expected, actual, error_ptr)
125         != SUCCESS)
126          returncode = FAILURE;
127  }
128  }
129  }
130  return(returncode);
131  }
132  }
133  }
134  /*
135  *   int vram_test(pam_no, error_ptr)
136  *   INPUT:  pam_no = PAM number
137  *           error_ptr = pointer to memory error structure
138  *   OUTPUT: returns SUCCESS or FAILURE
139  *   DESCRIPTION: Performs data bus test all video ram on specified PAM,
140  *                excluding parity ram.
141  */
142  int
143  vram_test(pam_no, error_ptr)
144  int pam_no;
145  MEM_ERRORS *error_ptr;
146  {
147      register int bank_no;
148      register u_long pam_base_address;
149      register u_long bank_base_address;
150      register int i;
151      int returncode = SUCCESS;
152
153      pam_base_address = Pam_base_address(current_lane, pam_no);
154      for(bank_no = 0; bank_no < 3; bank_no++)
155      {
156          bank_base_address = (u_long)get_base_address(pam_no, bank_no);
157          /* test the "even" and "odd" ICs in the bank */
158          for(i = 0; i < 4; i += 4)
159          {
160              if(data_bus_test(bank_base_address + i, error_ptr) != SUCCESS)
161              {
162                  returncode = FAILURE;
163                  if(!error_ptr) data_bus_test_fails.\n") != SUCCESS)
164                      return(FAILURE);
165              }
166          }
167      }
168      return(returncode);
169  }
170  }
171  }
172  /*
173  *   int vram_parity_test(pam_no, error_ptr)
174  *   INPUT:  pam_no = PAM number
175  *           error_ptr = pointer to memory error structure
176  *   OUTPUT: returns SUCCESS or FAILURE
177  *   DESCRIPTION: Performs data bus test on all parity video rams
178  *                on selected PAM
179  */
180  int
181  vram_parity_test(pam_no, error_ptr)
182  int pam_no;
183  MEM_ERRORS *error_ptr;
184  {
185      register int bank_no;
186      register int bit_pos;
187      register u_long pam_base_address;
188      register u_long ram_address;
189      register int i;
190      int returncode = SUCCESS;
191
192      Clear_pac_errors(current_lane);
193      pam_base_address = Pam_base_address(current_lane, pam_no);
194      /* test the four parity ICs on the board */
195      for(i = 0; i < 4; i += 2)
196      {
197          ram_address = pam_base_address + i;
198          /* Write and read walking bit in parity ram (bank selects bit i) */
199          for(bit_pos = 0; bit_pos < 3; bit_pos++)
200          {
201              /* write the bits */
202              for(bank_no = 0; bank_no < 3; bank_no++)
203              {
204                  Write_word(ram_address + (bank_no * PAT_MEM_BANK),
205                             (bank_no == bit_pos));
206              }
207              /* Read the bits and check for parity errors */
208              for(bank_no = 0; bank_no < 3; bank_no++)
209              {
210                  (void)Read_word(ram_address + (bank_no * PAT_MEM_BANK));
211                  if(pac_parity_error_check(error_ptr) != SUCCESS)
212                      returncode = FAILURE;
213              }
214          }
215      }
216      return(returncode);
217  }
218  }
219  }
220  /*
221  *   int pac_test_a_pam(pam_no)
222  *   INPUT:  pam_no = pattern memory to test (0,1,2,or3)
223  *   OUTPUT: returns SUCCESS or FAILURE
224  *   DESCRIPTION: Performs pattern memory test on specified PAM.
225  *                The procedure is as follows:
226  *   1) Perform an address line test
227  *   2) Initialize the error structure and set the high and low
228  *      parity generators to produce even parity.
229  *   3) Clear pattern memory.
230  *   4) Step through pattern memory, one bank at a time,
231  *      reading zeros and writing ones. This test checks
232  *      addressing and ability to clear all memory bits
233  *      (stored as 1's in VRAM's).
234  *   5) Step through pattern memory, one bank at a time,
235  *      reading ones and checking for parity errors after
236  *      each bank. This test checks ability to set all
237  *      memory bits (stored as 0's in VRAM's), and parity
238  *      VRAMs.
239  *   6) Now set the high and low parity generators to produce
240  */

```



Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM	DATE	PAGE #
	diags/pac_memtest.c	5/23/89	3/54
		TIME	4:41:19 pm

LINE #	SOURCE TEXT
241	/* odd parity (default). Then clear all of pattern memory.
242	/* This puts memory back in same state that it was in after
243	/* step 3, except the parity memory bits are inverted.
244	/* 5) Step through pattern memory, one bank at a time,
245	/* reading zeros and checking for parity errors after
246	/* each bank. This test checks opposite state of parity VRAMs.
247	/* 7) Now perform a data bus test on each VRAM. This entails
248	/* checking each bank, even and odd (6 locations), testing
249	/* eight ICs at once (4 bits each) by performing a walking
250	/* ones test followed by a walking zeros test.
251	/* 8) Finally, perform a walking ones/walking zeros test in
252	/* each parity VRAM.
253	/*
254	int
255	pac_test_s_pam(pam_no)
256	int pam_no;
257	{
258	MEM_ERRORS mem_errors; *err_ptr = &mem_errors;
259	int returncode = SUCCESS;
260	
261	/* Perform an address line test */
262	if(pam_addr_test(pam_no) != SUCCESS)
263	{
264	returncode = FAILURE;
265	if(!m_error("Address line test on PAM %d failed.\n", pam_no) != SUCCESS)
266	return FAILURE;
267	}
268	
269	/* Initialize the pattern memory error structure */
270	pac_init_mem_err(err_ptr);
271	
272	/* Set PAC for even parity */
273	Pac_set_parity(current_lase, SET_EVEN_PARITY);
274	
275	/* Clear pattern memory */
276	(void)pac_clear_s_pam(current_lase, pam_no);
277	
278	if(pac_read0s_writes(pam_no, err_ptr) != SUCCESS)
279	{
280	returncode = FAILURE;
281	if(!m_error("Pattern Memory %d fails \"read 0's, write 1's\" test.\n",
282	pam_no) != SUCCESS)
283	goto cleanup;
284	}
285	if(pac_read_value(pam_no, "01, err_ptr) != SUCCESS)
286	{
287	returncode = FAILURE;
288	if(!m_error("Pattern Memory %d fails \"read 1's\" test.\n", pam_no)
289	!= SUCCESS)
290	goto cleanup;
291	}
292	/* Set PAC back to odd parity */
293	Pac_set_parity(current_lase, SET_ODD_PARITY);
294	
295	/* Clear pattern memory */
296	(void)pac_clear_s_pam(current_lase, pam_no);
297	
298	if(pac_read_value(pam_no, "01, err_ptr) != SUCCESS)
299	{
300	returncode = FAILURE;
301	if(!m_error("Pattern Memory %d fails \"read 0's\" test.\n", pam_no)
302	!= SUCCESS)
303	goto cleanup;
304	}
305	if(vram_test(pam_no, err_ptr) != SUCCESS)
306	{
307	returncode = FAILURE;
308	if(!m_error("Pattern Memory %d data bus test fails.\n", pam_no)
309	!= SUCCESS)
310	goto cleanup;
311	}
312	/* See if parity errors occurred during the previous tests */
313	if(err_ptr->had_parity_errors == TRUE)
314	{
315	returncode = FAILURE;
316	(void)m_error("Parity errors occurred during memory tests.\n");
317	}
318	else if(returncode != FAILURE)
319	/* Test the parity bus bus if all previous tests have passed */
320	{
321	if(vram_parity_test(pam_no, err_ptr) != SUCCESS)
322	{
323	returncode = FAILURE;
324	(void)m_error("Pattern Memory %d parity bus test fails.\n", pam_no);
325	}
326	}
327	
328	cleanup:
329	if(returncode != SUCCESS)
330	{
331	
332	/* Restore odd parity and clear all PAC errors */
333	Pac_set_parity(current_lase, SET_ODD_PARITY);
334	Clear_pac_errors(current_lase);
335	pac_display_mem_err(err_ptr);
336	}
337	return(returncode);
338	}
339	
340	
341	/* Address line test */
342	pam_addr_test(pam_no)
343	int pam_no;
344	{
345	u_long *base;
346	register int bank;
347	register int bit;
348	register int index;
349	register int max_bits = bitcount(pac(current_lase).pam_size(pam_no));
350	register int high_offset = pac(current_lase).pam_size(pam_no) - 1;
351	int returncode = SUCCESS;
352	
353	/* Clear walking 1's and walking 0's address locations in each bank */
354	for (bank = 0; bank < 3; ++bank)
355	{
356	base = (u_long *)get_base_address(pam_no, bank);
357	for (bit = 0; bit < max_bits; ++bit)
358	{
359	index = 1 << bit;
360	base[index] = 0;

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_memtest.c

DATE 5/23/89  
TIME 4:41:19 pm

PAGE #  
4/55

```

LINE #          SOURCE TEXT
361      base[high_offset * index] = 01;
362      }
363      }
364      }
365      for (bank = 0; bank < 3; ++bank)
366      {
367          base = (u_long *)get_base_address(pam_no, bank);
368          /* Walking 1's */
369          /* Verify that values are cleared */
370          if(pac_verify_clear(pam_no, base, max_bits, 01, &returncode) != SUCCESS)
371              return(FAILURE);
372          base[0] = "01";
373          if (base[0] != "01") /* verify that base address was set */
374          {
375              returncode = FAILURE;
376              if(lm_error("PAM %d: Address %08X: Expected %08X, read %08X.\n",
377                  pam_no, base, "01", base[0]) != SUCCESS)
378                  return FAILURE;
379          }
380          /* Verify that values are still cleared */
381          if(pac_verify_clear(pam_no, base, max_bits, 01, &returncode) != SUCCESS)
382              return(FAILURE);
383          /* Walking 0's */
384          if(pac_verify_clear(pam_no, base, max_bits, high_offset, &returncode)
385              != SUCCESS)
386              return(FAILURE);
387          base[high_offset] = "01";
388          if (base[high_offset] != "01") /* verify that high address was set */
389          {
390              returncode = FAILURE;
391              if(lm_error("PAM %d: Address %08X: Expected %08X, read %08X.\n",
392                  pam_no, base[high_offset], "01", base[high_offset]) != SUCCESS)
393                  return FAILURE;
394          }
395          if(pac_verify_clear(pam_no, base, max_bits, high_offset, &returncode)
396              != SUCCESS)
397              return(FAILURE);
398          }
399      }
400      return(returncode);
401      }
402      }
403      int
404      pac_verify_clear(pam_no, base, max_bits, offset, returncode)
405      int pam_no;
406      u_long base;
407      int max_bits;
408      int offset;
409      int *returncode;
410      {
411          register int bit;
412          register int index;
413          for (bit = 0; bit < max_bits; ++bit)
414          {
415              index = offset * (1 << bit);
416              if (base[index] != 01)
417              {
418                  *returncode = FAILURE;
419                  if(lm_error("PAM %d: Address %08X: Expected 0, read %08X.\n",
420                      pam_no, base[index], base[index]) != SUCCESS)
421                      return(FAILURE);
422              }
423          }
424          return(SUCCESS);
425      }
426      }
427      int
428      get_base_address(board, bank)
429      int board;
430      int bank;
431      {
432          return Pattern_to_address(current_lase, bank,
433              pac_get_first_pattern_no(current_lase, board));
434      }
435      }
436      int
437      bitcount(x)
438      {
439          register int count;
440          for (count = 0; x = (x >> 1); ++count)
441          {
442              }
443          return(count);
444      }
445      }
446      /*
447      int pac_read0s_writals(pam_no, err_ptr)
448      *
449      * INPUT:  pam_no = PAM number
450      *          err_ptr = pointer to memory error structure
451      * OUTPUT: return SUCCESS or FAILURE
452      * DESCRIPTION: Performs test on pattern memory. Reads zeros
453      * and writes ones. The pattern memory must be xeroed before this
454      * routine is run.
455      */
456      int
457      pac_read0s_writals(pam_no, err_ptr)
458      int pam_no;
459      MEM_ERRORS *err_ptr;
460      {
461          register u_long *memptr;
462          register u_long temp;
463          register u_long i;
464          register int bank_no;
465          register int returncode = SUCCESS;
466          (void)lm_message("Reading 0's, writing 1's");
467          for(bank_no = 0; bank_no < 3; bank_no++)
468          {
469              lm_message("."); /* walking period before each bank */
470              memptr = (u_long *)get_base_address(pam_no, bank_no);
471              i = pac(current_lase).pam_size(pam_no);
472              do
473              {
474                  if((temp = *memptr) != 01)
475                  {
476                      returncode = FAILURE;
477                      if(pac_mem_error((u_long)memptr, 01, temp, err_ptr) != SUCCESS)
478                          return(returncode);
479                  }
480              }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_memtest.c	DATE 5/23/89	PAGE # 5/56
			TIME 4:41:19 pm	
LINE #	SOURCE TEXT			
481	*memptr++ = 01;			
482	} while (--i);			
483	}			
484	(void)lm_message("Done.\n");			
485	return(returncode);			
486	}			
487				
488	/*			
489	int pac_parity_error_check(err_ptr)			
490	/*			
491	INPUT: err_ptr = pointer to memory error structure			
492	OUTPUT: returns SUCCESS or FAILURE			
493	DESCRIPTION: Determines if a parity error was detected during			
494	a memory test read. If not, simply returns. If an error did			
495	occur, displays address and whether high word, low word, or			
496	both caused error. Fills in error structure and clears error.			
497	*/			
498	int			
499	pac_parity_error_check(err_ptr)			
500	MEM_ERRORS *err_ptr;			
501	{			
502	int returncode = SUCCESS;			
503	if(pacptr[current_lane] == high_word_parity_error == 1)			
504	{			
505	(void)lm_error("High word parity error reading address %08X.\n",			
506	Pac_parity_address(current_lane));			
507	returncode = FAILURE;			
508	}			
509	if(pacptr[current_lane] == low_word_parity_error == 1)			
510	{			
511	(void)lm_error("Low word parity error reading address %08X.\n",			
512	Pac_parity_address(current_lane));			
513	returncode = FAILURE;			
514	}			
515	if(returncode != SUCCESS)			
516	{			
517	/* Log into error structure and clear errors */			
518	err_ptr->had_parity_errors = TRUE;			
519	Clear_pac_errors(current_lane);			
520	}			
521	return(returncode);			
522	}			
523				
524	/*			
525	int pac_read_value(pam_no, value, err_ptr)			
526	/*			
527	INPUT: pam_no = PAM number			
528	value = 32-bit value to read			
529	err_ptr = pointer to memory error structure			
530	OUTPUT: returns SUCCESS or FAILURE			
531	DESCRIPTION: Performs test on pattern memory. Reads value			
532	specified. The pattern memory must be set to value before this			
533	routine is run.			
534	*/			
535	int			
536	pac_read_value(pam_no, value, err_ptr)			
537	int pam_no;			
538	register u_long value;			
539	MEM_ERRORS *err_ptr;			
540	{			
541	register u_long *memptr;			
542	register u_long temp;			
543	register u_long i;			
544	int returncode = SUCCESS;			
545	int bank_no;			
546	(void)lm_message("Reading %08X's", value);			
547	for(bank_no = 0; bank_no < 3; bank_no++)			
548	{			
549	lm_message(" "); /* walking period before each bank */			
550	memptr = (u_long *)get_base_address(pam_no, bank_no);			
551	i = pac[current_lane].pam_size(pam_no);			
552	bank_status = SUCCESS;			
553	do			
554	{			
555	if((temp = *memptr++) != value)			
556	{			
557	bank_status = FAILURE;			
558	if(pac_mem_error((u_long)(memptr - 1), value, temp, err_ptr) != SUCCESS)			
559	return(FAILURE);			
560	}			
561	} while (--i);			
562	if(bank_status == SUCCESS)			
563	{			
564	(void)pac_parity_error_check(err_ptr);			
565	}			
566	else			
567	{			
568	returncode = FAILURE;			
569	}			
570	(void)lm_message("Done.\n");			
571	return(returncode);			
572	}			
573	/*			
574	int pac_link_memtest()			
575	/*			
576	INPUT: none			
577	OUTPUT: returns SUCCESS or FAILURE			
578	DESCRIPTION: Performs link table memory test.			
579	*/			
580	int			
581	pac_link_memtest()			
582	{			
583	void pac_clear_link();			
584	register u_long *memptr;			
585	register u_long temp;			
586	register u_long i;			
587	u_long base_address;			
588	int returncode = SUCCESS;			
589	/* First clear link table */			
590	pac_clear_link();			
591	base_address = pac[current_lane].lane_offset + LINK_OFFSET;			
592	/* Next perform a read 0's, write 1's test */			
593	memptr = (u_long *)base_address;			
594	i = LINK_SIZE >> 2;			
595	do			
596	{			
597	if((temp = ((*memptr) & 0xffff)) != 01)			
598	{			
599	return(FAILURE);			
600	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_memtest.c	DATE 5/23/89	PAGE # 6/57
LINE #		SOURCE TEXT		
601		returncode = FAILURE;		
602		if(lm_error("Link table memory failure at address %08x.\n",		
603		\texpected = %04x, actual = %04x.\n", 0, memptr, temp) != SUCCESS)		
604		return(Failure);		
605		{		
606		*memptr++ = "01;		
607		} while (--i);		
608		/* Next perform a read 1's, write 0's test */		
609		memptr = (u_long *) (base_address);		
610		i = LINK_SIZE >> 2;		
611		do		
612		{		
613		if((temp = (*(memptr & 0xffff)) != 0xffff)		
614		{		
615		{		
616		returncode = FAILURE;		
617		if(lm_error("Link table memory failure at address %08x.\n",		
618		\texpected = %04x, actual = %04x.\n", 0xffff, memptr, temp) != SUCCESS)		
619		return(Failure);		
620		{		
621		*memptr++ = 01;		
622		} while (--i);		
623		/* Now perform a location test on the first link table location */		
624		if(location_test(base_address, 0, 16) != SUCCESS)		
625		{		
626		returncode = FAILURE;		
627		(void)lm_error("Link table data bus test fails.\n");		
628		{		
629		return(returncode);		
630		}		
631		}		
632		}		
633		/*		
634		link_pac_mem_error(address, expected, actual, mem_error_ptr)		
635		{		
636		/* INPUT: address = memory address of failed location		
637		expected = expected value of failed location		
638		actual = actual value of failed location		
639		mem_error_ptr = pointer to memory error structure		
640		/* OUTPUT: returns SUCCESS or FAILURE (return of lm_error)		
641		DESCRIPTION: Outputs error message based on arguments.		
642		Computes U-number of suspect memory IC(s). Fills in		
643		memory error structure;		
644		*/		
645		int		
646		pac_mem_error(address, expected, actual, mem_error_ptr)		
647		{		
648		register u_long address;		
649		register u_long expected;		
650		register u_long actual;		
651		MEM_ERRORS *mem_error_ptr;		
652		{		
653		register u_long ref_das;		
654		register u_long bad_bits;		
655		register int nibble;		
656		u_long base;		
657		char buffer[(4 * 8) + 1];		
658		char *bufptr = buffer;		
659		bad_bits = expected ^ actual;		
660		mem_error_ptr->bad_data_bits  = bad_bits;		
661		{		
662		base = 1 + ((2 - ((address >> 26) & 3)) << 1) + (1 - ((address >> 2) & 1));		
663		{		
664		for(nibble = 0; nibble < 8; nibble++)		
665		{		
666		if(((bad_bits >> (nibble << 2)) & 0xf) != 0)		
667		{		
668		ref_das = base + (26 * (1 - nibble / 4) + 6 + (3 - nibble & 4));		
669		/* Fill in memory error structure with ref_das */		
670		if(ref_das < 27)		
671		mem_error_ptr->vramalt0thru24  = 1 << ref_das;		
672		else		
673		mem_error_ptr->vramalt7thru50  = 1 << (ref_das - 26);		
674		/* Now fill up buffer for printing */		
675		*bufptr++ = '0';		
676		if(ref_das > 9)		
677		{		
678		*bufptr++ = '0' + (char)(ref_das / 10);		
679		*bufptr++ = '0' + (char)(ref_das % 10);		
680		{		
681		else		
682		*bufptr++ = '0' + (char)(ref_das);		
683		*bufptr++ = ' ';		
684		}		
685		*bufptr = '\0';		
686		{		
687		return(lm_error("Memory error: Addr=%08X Bits=%08X RAMS=%s\n",		
688		address, bad_bits, buffer));		
689		{		
690		}		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu.c

DATE 5/23/89  
TIME 4:41:20 pm

PAGE #  
1/58

```

LINE # SOURCE TEXT
1  /* SCCS ID: pac_menu.c rev 3.1, 4/24/89 at 07:49:19 */
2  /*****
3  *
4  *   pac_menu.c
5  *
6  *   Main Menu
7  *   used in Pattara Controller diagnostics
8  *
9  *****/
10 #include "common.h"
11 #include "lm_diags.h"
12 #include "mod_def.h"
13 #include "moduler_exta.h"
14 #include "pac_def.h"
15 #include "pac.h"
16 #include "pac_exta.h"
17
18 pac_diag_disp(parent_menu)
19 LM_DIAG_MENU *parent_menu;
20 {
21     int pac_reg_test();
22     int pac_link_mem_test();
23     int pac_idprom_test();
24     int pac_detect_pama();
25     int pac_check_pamid();
26     int pac_pat_mem_disp();
27     int pac_strapping_test();
28     int pac_cpu_parity_test();
29     int pac_pattern_disp();
30     int pac_util_disp();
31
32     static LM_DIAG_MENU_ITEM menu_list[] =
33     {
34         {
35             "1",
36             "Pattara Controller Register Test",
37             pac_reg_test,
38             LM_DIAG_diag_routine,
39             LM_DIAG_null
40         },
41         {
42             "2",
43             "Link Table Test",
44             pac_link_mem_test,
45             LM_DIAG_diag_routine,
46             LM_DIAG_null
47         },
48         {
49             "3",
50             "Pattara Controller ID Prom Test",
51             pac_idprom_test,
52             LM_DIAG_diag_routine,
53             LM_DIAG_null
54         },
55         {
56             "4",
57             "Pattara Memory Detection Test",
58             pac_detect_pama,
59             LM_DIAG_diag_routine,
60             LM_DIAG_null
61         },
62         {
63             "5",
64             "Pattara Memory ID Prom Test",
65             pac_check_pamid,
66             LM_DIAG_diag_routine,
67             LM_DIAG_null
68         },
69         {
70             "6",
71             "Pattara Memory Strapping Test",
72             pac_strapping_test,
73             LM_DIAG_diag_routine,
74             LM_DIAG_null
75         },
76         {
77             "7",
78             "Pattara Memory Test Menu",
79             pac_pat_mem_disp,
80             LM_DIAG_another_menu,
81             LM_DIAG_null
82         },
83         {
84             "8",
85             "Pattara Controller Parity Circuit Test",
86             pac_cpu_parity_test,
87             LM_DIAG_diag_routine,
88             LM_DIAG_null
89         },
90         {
91             "9",
92             "Pattara Play Test Menu",
93             pac_pattara_disp,
94             LM_DIAG_another_menu,
95             LM_DIAG_null
96         },
97         {
98             "10",
99             "Pattara Controller Utilities Menu",
100            pac_util_disp,
101            LM_DIAG_utility_menu,
102            LM_DIAG_null
103        }
104    };
105
106    static LM_DIAG_MENU pac_main_menu =
107    {
108        "PATTERN CONTROLLER DIAGNOSTICS",
109        sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
110        0,
111        menu_list
112    };
113
114    pac_main_menu.title = parent_menu->
115    menu_items[parent_menu->current_selection].menu_text;
116
117    return lm_display_menu(&pac_main_menu);
118 }
119
120 /*****
121 *   Pattara Memory Test Menu

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu.c

DATE 5/23/89  
TIME 4:41:20 pm

PAGE #  
2/59

```

LINE # SOURCE TEXT
-----
121 *****
122 pac_pat_mem_disp(parent_menu)
123 LM_DIAG_MENU "parent_menu:
124 {
125     register int pam_no;
126     static char pam_buffer[4][80];
127     int pac_pat_mem_test();
128     static LM_DIAG_MENU_ITEM menu_list[] =
129     {
130         {
131             "1",
132             pam_buffer[0],
133             pac_pat_mem_test,
134             LM_DIAG_diag_routine,
135             LM_DIAG_null,
136             "0"
137         },
138         {
139             "2",
140             pam_buffer[1],
141             pac_pat_mem_test,
142             LM_DIAG_diag_routine,
143             LM_DIAG_null,
144             "1"
145         },
146         {
147             "3",
148             pam_buffer[2],
149             pac_pat_mem_test,
150             LM_DIAG_diag_routine,
151             LM_DIAG_null,
152             "2"
153         },
154         {
155             "4",
156             pam_buffer[3],
157             pac_pat_mem_test,
158             LM_DIAG_diag_routine,
159             LM_DIAG_null,
160             "3"
161         }
162     },
163     static LM_DIAG_MENU menu =
164     {
165         0,
166         sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
167         0,
168         menu_list
169     },
170     menu.title = parent_menu->
171     menu.items[parent_menu->current_selection].menu_text,
172     for(pam_no = 0; pam_no < 4; pam_no++)
173     {
174         if(pac(current_lane).mem_pam > pam_no) /* enable test */
175         {
176             menu_list[pam_no].attributes |= LM_DIAG_disable;
177             sprintf(pam_buffer[pam_no], "PAM to Memory Test (tdk)", pam_no,
178                 pac(current_lane).pam_size[pam_no] >> 10);
179         }
180         else /* disable test */
181         {
182             menu_list[pam_no].attributes |= LM_DIAG_disable;
183             sprintf(pam_buffer[pam_no], "PAM to Memory Test ( - )", pam_no);
184         }
185     }
186     return lm_display_menu(&menu);
187 }
188
189 *****
190 " Pattern Play Menu
191 *****
192
193 #define MENU_INDEX_PAT_BIT 3
194 #define MENU_INDEX_PAT_BUS 4
195 #define MENU_INDEX_CRC 5
196
197 pac_patterns_disp(parent_menu)
198 LM_DIAG_MENU "parent_menu:
199 {
200     int pals_found;
201     int pac_freq_test();
202     int pac_error_test();
203     int pac_branch_test();
204     int pac_patterns_bits_test();
205     int pac_patterns_bus_test();
206     int pac_crc_test();
207     static LM_DIAG_MENU_ITEM menu_list[] =
208     {
209         {
210             "1",
211             "Frequency Sweep Test",
212             pac_freq_test,
213             LM_DIAG_diag_routine,
214             LM_DIAG_null
215         },
216         {
217             "2",
218             "Parity Error Test",
219             pac_error_test,
220             LM_DIAG_diag_routine,
221             LM_DIAG_null
222         },
223         {
224             "3",
225             "Fast Branching Test",
226             pac_branch_test,
227             LM_DIAG_diag_routine,
228             LM_DIAG_null
229         },
230         {
231             "4",
232             "Pattern Bit Test",
233             pac_patterns_bits_test,
234             LM_DIAG_diag_routine,
235             LM_DIAG_diag_routine
236         }
237     }
238 }
239
240

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu.c

DATE 5/23/89  
TIME 4:41:20 pm

PAGE #  
3/60

LINE # SOURCE TEXT

```

241 LM_DIAG_null
242 },
243 {
244     "5",
245     "Pattern Bus Test",
246     pac_pattern_bus_test,
247     LM_DIAG_diag_routine,
248     LM_DIAG_null
249 },
250 },
251 "6",
252 "Diagnostic Adapter CRC Test",
253 pac_crc_test,
254 LM_DIAG_diag_routine,
255 LM_DIAG_null
256 },
257 },
258 },
259 static LM_DIAG_MENU pac_play_menu =
260 {
261     "PATTERN PLAY SUB-MENU",
262     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
263     0,
264     menu_list
265 },
266 pac_play_menu.title = parent_menu->
267 menu_item[parent_menu->current_selection].menu_text;
268
269 if((pala_found = pac_check_for_pala()) == 0) /* No PELA in lane */
270 {
271     menu_list[MENU_INDEX_PAT_BIT].attributes |= LM_DIAG_disable;
272     menu_list[MENU_INDEX_PAT_BUS].attributes |= LM_DIAG_disable;
273     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
274 }
275 else
276 {
277     menu_list[MENU_INDEX_PAT_BIT].attributes |= LM_DIAG_disable;
278     menu_list[MENU_INDEX_PAT_BUS].attributes |= LM_DIAG_disable;
279     /* Check for diagnostic adapters */
280     if(pac_check_for_diag_dabs(pala_found) == 0) /* No diag dabs in lane */
281     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
282     else
283     menu_list[MENU_INDEX_CRC].attributes |= LM_DIAG_disable;
284 }
285
286 return la_display_menu(&pac_play_menu);
287 }
288
289
290 /*----- Pattern Controller Utilities Menu -----*/
291
292 pac_util_disp(parent_menu)
293 LM_DIAG_MENU *parent_menu;
294 {
295     int pac_config_report();
296     int pac_display_errors();
297     int pac_refresh_test();
298     int pac_fill_memory_disp();
299     int pac_play_util_disp();
300     int pac_sig_analysis();
301
302     static LM_DIAG_MENU_ITEM menu_list[] =
303     {
304         {
305             "1",
306             "Display Pattern Controller Configuration",
307             pac_config_report,
308             LM_DIAG_utility,
309             LM_DIAG_null
310         },
311         {
312             "2",
313             "Display Pattern Controller Errors",
314             pac_display_errors,
315             LM_DIAG_utility,
316             LM_DIAG_null
317         },
318         {
319             "3",
320             "Refresh Test",
321             pac_refresh_test,
322             LM_DIAG_utility,
323             LM_DIAG_null
324         },
325         {
326             "4",
327             "Fill Pattern Memory - Menu",
328             pac_fill_memory_disp,
329             LM_DIAG_utility_menu,
330             LM_DIAG_null
331         },
332         {
333             "5",
334             "Pattern Play - Menu",
335             pac_play_util_disp,
336             LM_DIAG_utility_menu,
337             LM_DIAG_null
338         },
339         {
340             "6",
341             "Signature Analysis",
342             pac_sig_analysis,
343             LM_DIAG_utility,
344             LM_DIAG_null
345         },
346     },
347     },
348 },
349 static LM_DIAG_MENU pac_util_menu =
350 {
351     "PATTERN CONTROLLER UTILITIES MENU",
352     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
353     0,
354     menu_list
355 },
356 pac_util_menu.title = parent_menu->
357 menu_item[parent_menu->current_selection].menu_text;
358
359 return la_display_menu(&pac_util_menu);
360 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_menu.c	DATE 5/23/89	PAGE # 4/61
			TIME 4:41:20 pm	

LINE #	SOURCE TEXT
361	.....
362	/*
363	Memory Fill Menu
364	.....*/
365	pac_fill_memory_disp(parent_menu)
366	LM_DIAG_MENU "parent_menu;
367	{
368	int pac_fill_mem_xero();
369	int pac_fill_mem_random();
370	int pac_fill_mem_counting();
371	int pac_fill_mem_walk1();
372	int pac_fill_mem_walk0();
373	int pac_fill_mem_lam00();
374	int pac_fill_mem_lam00();
375	static LM_DIAG_MENU_ITEM menu_list[] =
376	{
377	{
378	"1",
379	"Clear Pattern Memory",
380	pac_fill_mem_xero,
381	LM_DIAG_utility   LM_DIAG_automatic_quit,
382	LM_DIAG_null
383	},
384	{
385	"2",
386	"Fill with random data",
387	pac_fill_mem_random,
388	LM_DIAG_utility   LM_DIAG_automatic_quit,
389	LM_DIAG_null
390	},
391	{
392	"3",
393	"Fill with Counting Data",
394	pac_fill_mem_counting,
395	LM_DIAG_utility   LM_DIAG_automatic_quit,
396	LM_DIAG_null
397	},
398	{
399	"4",
400	"Fill with Walking Ones",
401	pac_fill_mem_walk1,
402	LM_DIAG_utility   LM_DIAG_automatic_quit,
403	LM_DIAG_null
404	},
405	{
406	"5",
407	"Fill with Walking Zeros",
408	pac_fill_mem_walk0,
409	LM_DIAG_utility   LM_DIAG_automatic_quit,
410	LM_DIAG_null
411	},
412	{
413	"6",
414	"Fill with Alternating Ones and Zeros",
415	pac_fill_mem_lam00,
416	LM_DIAG_utility   LM_DIAG_automatic_quit,
417	LM_DIAG_null
418	},
419	},
420	},
421	static LM_DIAG_MENU fill_memory_menu =
422	{
423	"FILL PATTERN MEMORY MENU",
424	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
425	0,
426	menu_list
427	},
428	fill_memory_menu.title = parent_menu->
429	menu_item(parent_menu->current_selection).menu_text,
430	return lm_display_menu(&fill_memory_menu);
431	}
432	.....
433	/*
434	Pattern Play Utility Menu
435	.....*/
436	pac_play_util_disp(parent_menu)
437	LM_DIAG_MENU "parent_menu;
438	{
439	int pac_sel_pat_clk();
440	int pac_single_play();
441	int pac_loop_with_sample();
442	int pac_loop_no_sample();
443	int pac_loop_no_pams();
444	static LM_DIAG_MENU_ITEM menu_list[] =
445	{
446	{
447	"1",
448	"Select Pattern Clock Frequency",
449	pac_sel_pat_clk,
450	LM_DIAG_utility,
451	LM_DIAG_null
452	},
453	{
454	"2",
455	"Single Presentation (reports time in ms)",
456	pac_single_play,
457	LM_DIAG_utility,
458	LM_DIAG_null
459	},
460	{
461	"3",
462	"Looping Play with Sample",
463	pac_loop_with_sample,
464	LM_DIAG_utility,
465	LM_DIAG_null
466	},
467	{
468	"4",
469	"Looping Play, no Sample",
470	pac_loop_no_sample,
471	LM_DIAG_utility,
472	LM_DIAG_null
473	},
474	{
475	"5",
476	"Looping Play, Dummy Pattern Memory",
477	pac_loop_no_pams,
478	LM_DIAG_utility,
479	LM_DIAG_null
480	},



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_menu.c	DATE 5/23/89	PAGE # 5/62
LINE #	SOURCE TEXT			
481	pac_loop_no_pams,			
482	lm_diag_utility,			
483	lm_diag_null			
484	}			
485	,			
486	static lm_diag_menu play_util_menu =			
487	{			
488	"PATTERN PLAY UTILITIES MENU",			
489	sizeof(menu_list) / sizeof(lm_diag_menu_item),			
490	0,			
491	menu_list			
492	};			
493	play_util_menu.title = parent_menu->			
494	menu_items(parent_menu->current_selection).menu_text,			
495	return lm_display_menu(&play_util_menu);			
496	}			
497				
498				

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_menu\_diag.c

DATE

5/23/89

PAGE #

TIME

4:41:20 pm

1/63

```

1  /* SCCS ID: pac_menu_diag.c rev 3.1, 4/24/89 at 07:49:23 */
2  /*
3  * .....
4  * pac_menu_diag.c
5  * .....
6  * Main diagnostic routines called directly by PAC menus
7  * used in PAC diagnostics
8  * .....
9  * .....
10 #include "vrtx.h" /* for in_delay and in_time */
11 #include "common.h"
12 #include "in_diags.h"
13 #include "mod_def.h"
14 #include "moder_exta.h"
15 #include "tag.h"
16 #include "tag_def.h"
17 #include "tag_exta.h"
18 #include "pac.h"
19 #include "magic.h"
20 #include "pel.h"
21 #include "pac_def.h"
22 #include "pac_exta.h"
23 #include "id.h"
24
25 /*
26  * int pac_error_test()
27  *
28  * INPUT: none
29  * OUTPUT: returns code = SUCCESS or FAILURE
30  * DESCRIPTION:
31  */
32 int
33 pac_error_test()
34 {
35     void pac_build_walking_branch();
36     int pattern;
37     int timeout;
38     u_long *memptr;
39     int branch_address;
40     int block_offset;
41     int gen_err;
42
43     if(pac_stack_pama(current_lane) != SUCCESS)
44     {
45         (void)lm_error("Pattern memory configuration error.\n");
46         return(FAILURE);
47     }
48
49     if(diag_clear_errors() != SUCCESS)
50         return(FAILURE);
51
52     /* Fill pattern memory with random data */
53     if(pac_fill_random(BLOCK_SIZE, (MAX_BRANCHES + 1) * BLOCK_SIZE, NOP_MASK,
54         SEED) != SUCCESS)
55     {
56         (void)lm_error("Could not fill with random data.\n");
57         return(FAILURE);
58     }
59
60     /* Set up pattern memory for "walking" branching */
61     pac_build_walking_branch();
62     if(pac_set_pattern_clock(PAC_MIN_PERIOD) != SUCCESS)
63     {
64         (void)lm_error("Could not set pattern clock to minimum period.\n");
65         return(FAILURE);
66     }
67
68     /* Setup good sample (PAC_MIN_PERIOD = 1000); */
69     /* Prepare for pattern play and compute a conservative timeout */
70     if((timeout = pac_pre_play()) == 0)
71     {
72         (void)lm_error("Pattern play preparation failed.\n");
73         return(FAILURE);
74     }
75
76     /* Put a stop instruction in the last block */
77     /* so that 10 patterns within the last block are played */
78     memptr = (u_long *) (pac(current_lane).lane_offset + BANK_2 +
79         ((MAX_BRANCHES + 1) * BLOCK_SIZE) + 10 * STOP_LATENCY);
80     *memptr = STOP;
81
82     (void)lm_message("Playing patterns");
83
84     for(pattern = BLOCK_SIZE, pattern < ((MAX_BRANCHES * BLOCK_SIZE) + 1);
85         pattern++)
86     {
87         if((pattern % (BLOCK_SIZE << 2)) == 0)
88         {
89             (void)lm_message("-");
90             gen_err = pac_predict_offsets(pattern, 5, &branch_address, &block_offset);
91             if(pac_insert_parity_error(pattern, 0) != SUCCESS)
92             {
93                 (void)lm_error("Could not insert parity error in pattern control.\n");
94                 return(FAILURE);
95             }
96
97             pacptr[current_lane]-->branch_address = 1;
98             if(pac_play(timeout) != SUCCESS)
99             {
100                 (void)lm_error("Pattern play failed.\n");
101                 return(FAILURE);
102             }
103
104             if(gen_err != SUCCESS)
105             {
106                 if(pacptr[current_lane]-->parity_error == TRUE)
107                 {
108                     (void)lm_error("Pattern play generated error.\n");
109                     return(FAILURE);
110                 }
111             }
112             else
113             {
114                 if(pacptr[current_lane]-->parity_error != TRUE)
115                 {
116                     (void)lm_error("Pattern play did not generate error (should have).\n");
117                     return(FAILURE);
118                 }
119                 if(pacptr[current_lane]-->branch_address != branch_address)
120                 {
121                     (void)lm_error("Branch address incorrect.\n\tExpected %d. Actual %d.\n",
122                         branch_address, pacptr[current_lane]-->branch_address);
123                     return(FAILURE);
124                 }
125                 if(pacptr[current_lane]-->block_offset != block_offset)
126                 {
127                     (void)lm_error("Block offset incorrect.\n\tExpected %d. Actual %d.\n",

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE 5/23/89  
TIME 4:41:20 pm

PAGE #  
2/64

```

121     block_offset, pacptr(current_lase) -> block_offset);
122     return(FAILURE);
123 }
124 }
125 if(pac_remove_parity_error(pattern, 0) != SUCCESS)
126 {
127     (void)lm_error("Could not remove parity error in pattern control.\n");
128     return(FAILURE);
129 }
130 if(gen_err != SUCCESS)
131 /* Increment pattern count to next block boundary (-1) */
132 pattern = (((pattern / BLOCK_SIZE) + 1) * BLOCK_SIZE) - 1;
133 }
134 (void)lm_message("done.\n");
135 return(SUCCESS);
136 }
137 }
138
139 /*
140  * int pac_cpu_parity_test()
141  *
142  * INPUT: none
143  * OUTPUT: return code = SUCCESS or FAILURE
144  * DESCRIPTION: Performs test on PAC parity
145  *               circuitry used for CPU reads and writes.
146  *               Tests are run on first location in pattern
147  *               memory.
148  */
149 int pac_cpu_parity_test()
150 {
151     if(pac_stack_pame(current_lase) != SUCCESS)
152     {
153         (void)lm_error("Pattern memory configuration error.\n");
154         return(FAILURE);
155     }
156     /* Perform pac_parity_test using first location in Bank 0 of pattern mem */
157     return pac_parity_test(pac(current_lase).lase_offset);
158 }
159
160
161 /*
162  * int pac_refresh_test()
163  *
164  * INPUT: none
165  * OUTPUT: return code = SUCCESS or FAILURE
166  * DESCRIPTION: Tests refresh of pattern memory
167  *               by writing random data, pausing for 1 second,
168  *               and then reading the data back. This is repeated
169  *               for a user specified number of trials. Each trial
170  *               uses a new seed (random) to generate a different
171  *               set of random numbers.
172  */
173 int pac_refresh_test()
174 {
175     u_long random_seed;
176     u_long i;
177     u_long trials;
178     u_long failures;
179
180     if(diag_clear_errors() != SUCCESS)
181         return(FAILURE);
182
183     failures = 0;
184     random_seed = SEED;
185     trials = 1;
186     diag_get_ulong(&trials, "number of trials", 11, 65535);
187     (void)lm_message("Hit key to exit after a trial.\n");
188     for(i = 0; i < trials; i++)
189     {
190         /* Fill all of pattern memory with random data, using random seed */
191         if(pac_fill_random(0, pac(current_lase).num_patterns, 01,
192             (random_seed = pac_get_random(random_seed))) != SUCCESS)
193         {
194             (void)lm_error("Could not perform refresh test.\n");
195             return(FAILURE);
196         }
197         lm_delay(1000); /* Pause for 1 second */
198         if(pac_read_random(0, pac(current_lase).num_patterns, "01, random_seed")
199             != SUCCESS)
200         {
201             if(lm_error("Refresh test fails random read number %d.\n", i + 1)
202                 != SUCCESS)
203                 return(FAILURE);
204             else
205                 ++failures;
206         }
207         else
208             (void)lm_message("Trial number %d passes.\n", i + 1);
209     }
210     clear_key_buf();
211     if(failures != 0)
212     {
213         (void)lm_error("Refresh test failed %d times.\n", failures);
214         return(FAILURE);
215     }
216     else
217         (void)lm_message("Refresh test passes.\n");
218     return(SUCCESS);
219 }
220
221
222 /*
223  * int pac_branch_test()
224  *
225  * INPUT: none
226  * OUTPUT: return code = SUCCESS or FAILURE
227  * DESCRIPTION:
228  */
229 int pac_branch_test()
230 {
231     void pac_build_fast_branch();
232     int patterns;
233     int returncode = SUCCESS;
234
235     if(pac_stack_pame(current_lase) != SUCCESS)
236     {
237         (void)lm_error("Pattern memory configuration error.\n");
238     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:20 pm	3/65

```

LINE #          SOURCE TEXT
241      return(FAILURE);
242  }
243
244  if(diag_clear_errors() != SUCCESS)
245      return(FAILURE);
246
247  patterns = pac(current_lane).num_blocks * (BRANCH_LATENCY + 1);
248  /* Fill all of pattern memory with random data */
249  if(pac_fill_random(0, pac(current_lane).num_patterns, NOP_MASK, SEED)
250     != SUCCESS)
251  {
252      (void)lm_error("Could not perform branch test.\n");
253      return(FAILURE);
254  }
255  /* Set up pattern memory for fast branching */
256  pac_build_fast_branch();
257  if(pac_sweep_test(patterns) != SUCCESS)
258  {
259      returncode = FAILURE;
260      if(lm_error("Fast branching, random data, test failed.\n") != SUCCESS)
261          return(FAILURE);
262  }
263
264  /* Fill all of pattern memory with alternating ones and zeros data */
265  if(pac_fill_1and0(0, pac(current_lane).num_patterns, NOP_MASK) != SUCCESS)
266  {
267      (void)lm_error("Could not perform branch test.\n");
268      return(FAILURE);
269  }
270  /* Set up pattern memory for fast branching */
271  pac_build_fast_branch();
272  if(pac_sweep_test(patterns) != SUCCESS)
273  {
274      (void)lm_error("Fast branching, 1/0 data, test failed.\n");
275      return(FAILURE);
276  }
277  return(returncode);
278  }
279
280  /*
281   *   int pac_single_play()
282   *
283   *   INPUT: none
284   *   OUTPUT: return code = SUCCESS or FAILURE
285   *   DESCRIPTION:
286   */
287  int
288  pac_single_play()
289  {
290      int sample_width;
291      int first_patterns;
292      int patterns;
293      int play_time;
294      int period;
295
296      if(diag_clear_errors() != SUCCESS)
297          return(FAILURE);
298
299      sample_width = 16;
300      diag_get_long(&(long)sample_width, "sample width", 01, 2551);
301
302      first_patterns = 0;
303      diag_get_long(&(long)first_patterns, "first pattern number", 01,
304                  (long)(pac(current_lane).num_patterns - BLOCK_SIZE));
305
306      patterns = pac(current_lane).num_patterns;
307      diag_get_long(&(long)patterns, "number of patterns", 01, (long)patterns);
308
309      if(build_patterns_control(first_patterns, patterns, STOP_MODE) != SUCCESS)
310      {
311          (void)lm_error("Unable to build pattern control.\n");
312          return(FAILURE);
313      }
314      /* Insert PFI control in last pattern to generate sample */
315      *((u_long *) (pac(current_lane).lane_offset + BANK_2 * 4 * (first_patterns +
316      patterns - 1))) |= 0x7 << 4;
317      sample_width = sample_width;
318      /* Prepare for pattern play and compute a conservative timeout */
319      if(pac_pre_play() == 0)
320      {
321          (void)lm_error("Pattern play preparation failed.\n");
322          return(FAILURE);
323      }
324
325      period = tny_measure_period();
326      setup_good_sample(period);
327      if(pac_timed_play(period / 1000, patterns, splay_time)
328         != SUCCESS)
329      {
330          (void)lm_error("Timed play fails.\n");
331          return(FAILURE);
332      }
333      (void)lm_message("Play lasted %dms +/- %dms.\n", play_time, TIMER_RES);
334      return(SUCCESS);
335  }
336
337  /*
338   *   int pac_loop_no_sample()
339   *
340   *   INPUT: none
341   *   OUTPUT: return code = SUCCESS or FAILURE
342   *   DESCRIPTION:
343   */
344  int
345  pac_loop_no_sample()
346  {
347      int first_patterns;
348      int patterns;
349      int period;
350
351      if(diag_clear_errors() != SUCCESS)
352          return(FAILURE);
353
354      /* Set sample width to 16 */
355      tnypr->sample_width = 16;
356
357      first_patterns = 0;
358      diag_get_long(&(long)first_patterns, "first pattern number", 01,
359                  (long)(pac(current_lane).num_patterns - BLOCK_SIZE));
360

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_menu_diag.c	DATE 5/23/89	PAGE # 4/66
LINE #	SOURCE TEXT			
361	patterns = pac(current_lane).num_patterns;			
362	diag_get_long(&(long)patterns, "number of patterns", 41, (long)patterns);			
363				
364	if(build_pattern_control(first_pattern, patterns, LOOP_MODE) != SUCCESS)			
365	{			
366	(void)lm_error("Unable to build pattern con. . .\n");			
367	return(FAILURE);			
368	}			
369	/* Prepare for pattern play and compute a conservative timeout */			
370	if(pac_pre_play() == 0)			
371	{			
372	(void)lm_error("Pattern play preparation failed.\n");			
373	return(FAILURE);			
374	}			
375				
376	period = tmg_measure_period();			
377	setup_good_sample(period);			
378				
379	(void)lm_message("Initiating looping play...\n");			
380	(void)lm_message("Hit key to abort.\n");			
381	if(tmg_initiate_play() != SUCCESS)			
382	{			
383	(void)lm_error("Unable to initiate play.\n");			
384	pac_play_cleanup();			
385	return(FAILURE);			
386	}			
387	/* Wait for key hit */			
388	(void)lm_get_key();			
389	Clear_key_buf();			
390	if(Get_bp_error() == Lane_code(current_lane))			
391	{			
392	(void)lm_error("Error line asserted on backplane.\n");			
393	report_bp_error();			
394	return(FAILURE);			
395	}			
396	else			
397	{			
398	if(pac_abort_play() != SUCCESS)			
399	{			
400	(void)lm_error("Could not abort play.\n");			
401	return(FAILURE);			
402	}			
403	}			
404	else			
405	{			
406	(void)lm_message("Looping aborted.\n");			
407	return(SUCCESS);			
408	}			
409	}			
410				
411				
412	/*			
413	int pac_link_mem_test()			
414	{			
415	INPUT: none			
416	OUTPUT: return code = SUCCESS or FAILURE			
417	DESCRIPTION: Tests link table memory. The			
418	link table is a 16-bit memory on 32-bit			
419	boundaries.			
420	}			
421	int			
422	pac_link_mem_test()			
423	{			
424	if(pac_link_memtest() != SUCCESS)			
425	return(FAILURE);			
426	return(SUCCESS);			
427	}			
428				
429	/*			
430	int pac_pat_mem_test(status, info)			
431	{			
432	INPUT: status = address of status word			
433	info = address of test info; in this case, PAM number			
434	OUTPUT: return code = SUCCESS or FAILURE			
435	DESCRIPTION: Tests pattern memory on specified PAM			
436	}			
437	int			
438	pac_pat_mem_test(status, info)			
439	{			
440	char *info;			
441	if(pac_stack_pams(current_lane) != SUCCESS)			
442	{			
443	(void)lm_error("Pattern memory configuration error.\n");			
444	return(FAILURE);			
445	}			
446				
447	if(diag_clear_errors() != SUCCESS)			
448	return(FAILURE);			
449				
450	return(pac_test_e_pam((int)('info' - '0')));			
451	}			
452				
453				
454	/*			
455	int pac_freq_test()			
456	{			
457	INPUT: none			
458	OUTPUT: return code = SUCCESS or FAILURE			
459	DESCRIPTION:			
460	}			
461	int			
462	pac_freq_test()			
463	{			
464	register int pam_no;			
465	register int first_pattern;			
466	register int num_patterns;			
467	int returncode = SUCCESS;			
468				
469	if(pac_stack_pams(current_lane) != SUCCESS)			
470	{			
471	(void)lm_error("Pattern memory configuration error.\n");			
472	return(FAILURE);			
473	}			
474	if(diag_clear_errors() != SUCCESS)			
475	return(FAILURE);			
476				
477	for(pam_no = 0; pam_no < pac(current_lane).num_pams; pam_no++)			
478	{			
479	first_pattern = pac_get_first_pattern_no(current_lane, pam_no);			
480	num_patterns = pac(current_lane).pam_size[pam_no];			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:20 pm	5/67

```

LINE #          SOURCE TEXT
481  /* Fill pattern memory with random data */
482  if(pac_fill_random(first_patterns, num_patterns, MOP_MASK, SEED) != SUCCESS)
483  {
484      returncode = FAILURE;
485      if(!m_error("Could not perform frequency test on PAM ad.\n", pam_bo)
486         != SUCCESS)
487          return(FAILURE);
488      else
489          continue;
490  }
491  /* Set up pattern memory to branch through pattern memory */
492  if(build_pattern_control(first_patterns, num_patterns, STOP_MODE) != SUCCESS)
493  {
494      returncode = FAILURE;
495      if(!m_error("Could not perform frequency test on PAM ad.\n", pam_bo)
496         != SUCCESS)
497          return(FAILURE);
498      else
499          continue;
500  }
501  if(pac_sweep_test(num_patterns) != SUCCESS)
502  {
503      returncode = FAILURE;
504      if(!m_error("Frequency sweep test failed using PAM ad.\n", pam_bo)
505         != SUCCESS)
506          return(FAILURE);
507      else
508          continue;
509  }
510  }
511  return(returncode);
512 }
513
514
515
516 /*
517  * int pac_fill_mem_random()
518  * INPUT: none
519  * OUTPUT: return code = SUCCESS or FAILURE
520  * DESCRIPTION: fills entire pattern memory with
521  * a pseudorandom sequence.
522  */
523 int
524 pac_fill_mem_random()
525 {
526     if(pac_fill_random(0, pac(current_lane).num_patterns, "01, SEED") != SUCCESS)
527     {
528         (void)m_error("Could not fill pattern memory.\n");
529         return(FAILURE);
530     }
531     return(SUCCESS);
532 }
533
534 /*
535  * int pac_fill_mem_counting()
536  * INPUT: none
537  * OUTPUT: return code = SUCCESS or FAILURE
538  * DESCRIPTION: fills entire pattern memory with
539  * counting data.
540  */
541 int
542 pac_fill_mem_counting()
543 {
544     if(pac_fill_counting(0, pac(current_lane).num_patterns, "01") != SUCCESS)
545     {
546         (void)m_error("Could not fill pattern memory.\n");
547         return(FAILURE);
548     }
549     return(SUCCESS);
550 }
551
552 /*
553  * int pac_fill_mem_walking()
554  * INPUT: none
555  * OUTPUT: return code = SUCCESS or FAILURE
556  * DESCRIPTION: fills entire pattern memory with
557  * walking ones data.
558  */
559 int
560 pac_fill_mem_walking()
561 {
562     if(pac_fill_walking(0, pac(current_lane).num_patterns, "01, 1") != SUCCESS)
563     {
564         (void)m_error("Could not fill pattern memory.\n");
565         return(FAILURE);
566     }
567     return(SUCCESS);
568 }
569
570 /*
571  * int pac_fill_mem_walking0()
572  * INPUT: none
573  * OUTPUT: return code = SUCCESS or FAILURE
574  * DESCRIPTION: fills entire pattern memory with
575  * walking zeros data.
576  */
577 int
578 pac_fill_mem_walking0()
579 {
580     if(pac_fill_walking(0, pac(current_lane).num_patterns, "01, 0") != SUCCESS)
581     {
582         (void)m_error("Could not fill pattern memory.\n");
583         return(FAILURE);
584     }
585     return(SUCCESS);
586 }
587
588 /*
589  * int pac_fill_mem_land0()
590  * INPUT: none
591  * OUTPUT: return code = SUCCESS or FAILURE
592  * DESCRIPTION: fills entire pattern memory with
593  * alternating ones and zeros data.
594  */
595 int
596 pac_fill_mem_land0()
597 {
598     if(pac_fill_land0(0, pac(current_lane).num_patterns, "01") != SUCCESS)
599     {
600

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:20 pm 6/68

```

LINE # SOURCE TEXT
601 {
602     (void)lm_error("Could not fill pattern memory.\n");
603     return(FAILURE);
604 }
605 return(SUCCESS);
606 }
607
608
609 /*
610  * int pac_detect_pams(istatus)
611  *
612  * INPUT: istatus = address of status word
613  * OUTPUT: return code = SUCCESS or FAILURE
614  * DESCRIPTION: Checks to see if PAMs can be tested
615  *             detected on current PAC. If not, tests are aborted.
616  */
617 int
618 pac_detect_pams(istatus)
619 u_long *status;
620 {
621     if(pac_get_num_pams(current_lane) != SUCCESS)
622     {
623         (void)lm_error("PAM detect test failed. Cannot perform further tests \
624 on PAC.\n");
625         STOP_TEST(status);
626         return(FAILURE);
627     }
628     return(SUCCESS);
629 }
630
631 /*
632  * int pac_check_pamid(istatus)
633  *
634  * INPUT: istatus = address of status word
635  * OUTPUT: return code = SUCCESS or FAILURE
636  * DESCRIPTION: Checks the PAM ID Proms. If any
637  *             fail checksum, tests are aborted.
638  */
639 int
640 pac_check_pamid(istatus)
641 u_long *status;
642 {
643     static char message[] =
644         "PAM ID Prom test failed. Cannot perform further tests on PAC.\n";
645     if(pac(current_lane).num_pams == 0)
646     {
647         (void)lm_error("No pattern memory in this lane.\n");
648         return FAILURE;
649     }
650     if(pam_idprom_test(current_lane) != SUCCESS)
651     {
652         (void)lm_error(message);
653         STOP_TEST(status);
654         return(FAILURE);
655     }
656     return(SUCCESS);
657 }
658
659
660
661
662 pac_strapping_test(status)
663 u_long *status;
664 {
665     if(pac_stack_pams(current_lane) != SUCCESS)
666     {
667         (void)lm_error("Pattern memory configuration error.\n");
668         STOP_TEST(status);
669         return(FAILURE);
670     }
671     return SUCCESS;
672 }
673
674
675 /*
676  * int pac_fill_mem_zero()
677  *
678  * INPUT: none
679  * OUTPUT: return code = SUCCESS or FAILURE
680  * DESCRIPTION: Clears pattern memory.
681  */
682 int
683 pac_fill_mem_zero()
684 {
685     /* Check to make sure that there are PAMs on the PAC */
686     if(pac(current_lane).num_pams == 0)
687     {
688         (void)lm_error("There are no PAMs on this PAC.\n");
689         return(FAILURE);
690     }
691     else
692     {
693         (void)pac_clear_pat_mem(current_lane);
694         return(SUCCESS);
695     }
696 }
697
698 /*
699  * int pac_sel_pat_clk()
700  *
701  * INPUT: none
702  * OUTPUT: return code = SUCCESS or FAILURE
703  * DESCRIPTION: Sets pattern clock frequency by getting
704  *             the desired clock period from the keyboard. Also allows
705  *             the user to select an external clock. The clock is not
706  *             turned on by the function.
707  */
708 int
709 pac_sel_pat_clk()
710 {
711     int user_input;
712     (void)lm_message("Enter 0 for external clock 0.\n");
713     (void)lm_message("enter 1 for external clock 1.\n");
714     (void)lm_message("or enter period for internal clock.\n");
715     (void)lm_message("(id <= period <= 1d)\n", PAC_MIN_PERIOD, PAC_MAX_PERIOD);
716     user_input = 401; /* 25 MHz */
717     diag_get_long(&long)user_input, "value", 01, (long)PAC_MAX_PERIOD);
718     if(user_input == 0)
719     {
720         tmpptr->clock_select = EXT_CLOCK_0;
721         (void)lm_message("External clock 0 selected.\n");
722     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:20 pm	7/69

LINE #	SOURCE TEXT
721	else if(user_input == 1)
722	{
723	tmpstr->clock_select = EXT_CLOCK_1;
724	(void)lm_message("External clock 1 selected.\n");
725	}
726	else
727	{
728	if(pac_set_pattern_clock(user_input) != SUCCESS)
729	{
730	(void)lm_error("Unable to set internal clock frequency.\n");
731	return(FAILURE);
732	}
733	else
734	{
735	/* Make sure PAC clock speed register set properly */
736	(void)pac_pre_play();
737	}
738	return(SUCCESS);
739	}
740	
741	
742	
743	
744	/*
745	int pac_idprom_test()
746	{
747	INPUT: none
748	OUTPUT: return code = SUCCESS or FAILURE
749	DESCRIPTION: Performs checksum test on PAC ID PROM.
750	The function returns SUCCESS if the checksum is
751	correct, and returns FAILURE if the checksum test fails.
752	}
753	int
754	pac_idprom_test()
755	{
756	int tmp;
757	if((tmp = id_checksum((u_char *) (pac[current_lane].lane_offset +
758	PAC_ID + 3))) != ID_CHECKSUM_GOOD)
759	{
760	if(lm_error("PAC ID PROM checksum. Expected = 002x. Actual = 002x.\n",
761	ID_CHECKSUM_GOOD, tmp) != SUCCESS)
762	return(FAILURE);
763	}
764	return(SUCCESS);
765	}
766	
767	
768	/*
769	int pac_reg_test()
770	{
771	INPUT: none
772	OUTPUT: return code = SUCCESS or FAILURE
773	DESCRIPTION: Performs register test on all PAC read/write
774	registers. Returns SUCCESS or FAILURE.
775	}
776	int
777	pac_reg_test()
778	{
779	u_long addr;
780	u_long state;
781	int returncode = SUCCESS;
782	/* Test the PAC Configuration Register */
783	addr = pac[current_lane].lane_offset + CONFIG_OFFSET;
784	state = Read_long(addr); /* Save the state of the register */
785	if(location_test(addr, 0, 8) != SUCCESS)
786	{
787	returncode = FAILURE;
788	if(lm_error("PAC Configuration Register test fails.\n") != SUCCESS)
789	return(FAILURE);
790	}
791	Write_long(addr, state); /* Return register to former state */
792	/* Test the PAC Clock Speed Register */
793	addr = pac[current_lane].lane_offset + CLOCK_OFFSET;
794	state = Read_long(addr); /* Save the state of the register */
795	if(location_test(addr, 0, 1) != SUCCESS)
796	{
797	returncode = FAILURE;
798	if(lm_error("PAC Clock Speed Register test fails.\n") != SUCCESS)
799	return(FAILURE);
800	}
801	Write_long(addr, state); /* Return register to former state */
802	/* Test the PAC Branch Address Register */
803	addr = pac[current_lane].lane_offset + BRANCH_OFFSET;
804	state = Read_long(addr); /* Save the state of the register */
805	if(location_test(addr, 0, 16) != SUCCESS)
806	{
807	returncode = FAILURE;
808	if(lm_error("PAC Branch Address Register test fails.\n") != SUCCESS)
809	return(FAILURE);
810	}
811	Write_long(addr, state); /* Return register to former state */
812	return(returncode);
813	}
814	
815	
816	
817	
818	int
819	pac_config_report()
820	{
821	int i;
822	if(pac_stack_pams(current_lane) != SUCCESS)
823	{
824	(void)lm_error("PAC configuration error.\n");
825	return(FAILURE);
826	}
827	else
828	{
829	(void)lm_message("PAC configuration successful.\n");
830	(void)lm_message("There are a total of %d PAMs.\n",
831	pac[current_lane].num_pams);
832	for(i=0; i < pac[current_lane].num_pams; i++)
833	{
834	(void)lm_message(" PAM %d is a %dk PAM.\n", i,
835	pac[current_lane].pam_size[i] >> 10);
836	}
837	(void)lm_message("There are a total of %dk patterns (%d blocks).\n",
838	pac[current_lane].num_patterns >> 10, pac[current_lane].num_blocks);
839	}
840	return(SUCCESS);



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE 5/23/89  
TIME 4:41:20 pm

PAGE #  
8/70

```

LINE # SOURCE TEXT
841 }
842 }
843
844 int
845 pac_check_errors(lane, pptr)
846 int lane;
847 int (*ptrfn)();
848 {
849     int returncode = SUCCESS;
850
851     if((pacptr[lane]-->refresh_error) != 0)
852     {
853         returncode = FAILURE;
854         (void)ptrfn("Pattern Controller Refresh error.\n");
855     }
856     if((pacptr[lane]-->request_error) != 0)
857     {
858         returncode = FAILURE;
859         (void)ptrfn("Pattern Controller Request Control Machine error.\n");
860     }
861     if((pacptr[lane]-->pattern_error) != 0)
862     {
863         returncode = FAILURE;
864         (void)ptrfn("Pattern Control Machine error.\n");
865     }
866     if((pacptr[lane]-->parity_error) != 0)
867     {
868         returncode = FAILURE;
869         (void)ptrfn("Pattern Controller pattern control word parity error.\n");
870         block_number = 1d.\n";
871         block_offset = 1d.\n"; pacptr[lane]-->branch_address,
872         pacptr[lane]-->block_offset);
873     }
874     if((pacptr[lane]-->high_word_parity_error) != 0)
875     {
876         returncode = FAILURE;
877         (void)ptrfn("Pattern Controller High Word parity error.\n");
878     }
879     if((pacptr[lane]-->low_word_parity_error) != 0)
880     {
881         returncode = FAILURE;
882         (void)ptrfn("Pattern Controller Low Word parity error.\n");
883     }
884     if((pacptr[lane]-->low_word_parity_error) != 0)
885     {
886         if((pacptr[lane]-->high_word_parity_error) != 0)
887         {
888             (void)ptrfn("Pattern Controller parity error address = 108X.\n");
889             Pac_parity_address(lane);
890             return(returncode);
891         }
892     }
893
894     int
895     pac_display_errors()
896     {
897         if(pac_check_errors(current_lane, lm_message) == SUCCESS)
898             (void)lm_message("No Pattern Controller error conditions present.\n");
899         return(SUCCESS);
900     }
901
902     /*
903     int pac_loop_no_pams()
904     *
905     INPUT: none
906     OUTPUT: return code = SUCCESS or FAILURE
907     DESCRIPTION:
908     */
909     int
910     pac_loop_no_pams()
911     {
912         int timeout;
913         int period;
914
915         /* Check to make sure that there are no PAMs on the PAC */
916         if(pac(current_lane).num_pams != 0)
917         {
918             (void)lm_error("There are PAMs on this PAC.\n");
919             return(FAILURE);
920         }
921
922         Clear_pac_errors(current_lane);
923
924         pacptr[current_lane]-->low_word_parity = SET_ODD_PARITY;
925         tmgptr-->sample_width = 16;
926
927         /* Prepare for pattern play and compute a conservative timeout */
928         if((timeout = pac_pre_play()) == 0)
929         {
930             (void)lm_error("Pattern play preparation failed.\n");
931             return(FAILURE);
932         }
933         period = tmg_measure_period();
934         setup_good_sample(period);
935         (void)lm_message("Initiating looping play...\n");
936         (void)lm_message("Hit key to abort.\n");
937         while((Get_bp_error() != Lane_code(current_lane)) && (lm_check_key() == 0x0))
938         {
939             pacptr[current_lane]-->branch_address = 0;
940             if(pac_play(timeout) != SUCCESS)
941             {
942                 pac_play_cleanup();
943                 break;
944             }
945         }
946         Clear_key_buf();
947         if(Get_bp_error() == Lane_code(current_lane))
948         {
949             (void)lm_error("Error line asserted on backplane.\n");
950             (void)report_bp_error();
951         }
952         else
953             (void)lm_message("Looping aborted.\n");
954         return(SUCCESS);
955     }
956
957     /*
958     int pac_loop_with_sample()
959     *
960     INPUT: none
961     OUTPUT: return code = SUCCESS or FAILURE
962     DESCRIPTION:
963     */

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:20 pm

9/71

```

LINE # SOURCE TEXT
961 int
962 pac_loop_with_sample()
963 {
964     long sample_width;
965     int first_pattern;
966     int first_block;
967     int patterns;
968     int timeout;
969     int period;
970
971     if(diag_clear_errors() != SUCCESS)
972         return(FAILURE);
973
974     sample_width = 16;
975     diag_get_log(&sample_width, "sample width", 01, 2551);
976     first_pattern = 0;
977     diag_get_log(&(long)first_pattern, "first pattern number", 01,
978     (long)(pac(current_lane).sum_patterns - BLOCK_SIZE));
979     patterns = pac(current_lane).sum_patterns;
980     diag_get_log(&(long)patterns, "number of patterns", 41, (long)patterns);
981
982     if(build_patterns_control(first_pattern, patterns, STOP_MODE) != SUCCESS)
983     {
984         (void)lm_error("Unable to build patterns control.\n");
985         return(FAILURE);
986     }
987     /* Insert PEL control in last pattern to generate sample */
988     *(u_long *)(&pac(current_lane).lane_offset + BANK_2 + 4 * (first_pattern +
989     patterns - 1)) |= 0x7 << 4;
990     trigger->sample_width = sample_width;
991     first_block = first_pattern/BLOCK_SIZE;
992
993     /* Prepare for patterns play and compute a conservative timeout */
994     if((timeout = pac_pre_play()) == 0)
995     {
996         (void)lm_error("Patterns play preparation failed.\n");
997         return(FAILURE);
998     }
999     period = tmg_measure_period();
1000     setup_good_sample(period);
1001     (void)lm_message("Initiating looping play...\n");
1002     (void)lm_message("Hit key to abort.\n");
1003     while((Get_bp_error() != Lane_code(current_lane)) && (lm_check_key() == 0x0))
1004     {
1005         pacptr(current_lane)->branch_address = first_block;
1006         if(pac_play(timeout) != SUCCESS)
1007         {
1008             pac_play_cleanup();
1009             break;
1010         }
1011     }
1012     Clear_key_buf();
1013     if(Get_bp_error() == Lane_code(current_lane))
1014     {
1015         (void)lm_error("Error line asserted on backplane.\n");
1016         (void)report_bp_error();
1017     }
1018     else
1019     {
1020         (void)lm_message("Looping aborted.\n");
1021         return(SUCCESS);
1022     }
1023
1024     /*
1025     int pac_sig_analysis()
1026     {
1027         INPUT: none
1028         OUTPUT: return code = SUCCESS or FAILURE
1029         DESCRIP: On: Parts lane is signature analysis
1030         mode, playing random data through entire memory.
1031         The trigger pin is set in the first patterns only.
1032     */
1033     int
1034     pac_sig_analysis()
1035     {
1036         if(diag_clear_errors() != SUCCESS)
1037             return(FAILURE);
1038
1039         if(pac_fill_random(0, pac(current_lane).sum_patterns, "0x0801, SEED")
1040         != SUCCESS)
1041         {
1042             (void)lm_error("Could not fill patterns memory.\n");
1043             return(FAILURE);
1044         }
1045         if(build_patterns_control(0, pac(current_lane).sum_patterns, LOOP_MODE)
1046         != SUCCESS)
1047         {
1048             (void)lm_error("Unable to build patterns control.\n");
1049             return(FAILURE);
1050         }
1051         /* Set trigger bit in first location of patterns memory */
1052         *(u_long *)(&pac(current_lane).lane_offset + BANK_2) |= 0x0801;
1053         /* Prepare for patterns play and compute a conservative timeout */
1054         if(pac_pre_play() == 0)
1055         {
1056             (void)lm_error("Patterns play preparation failed.\n");
1057             return(FAILURE);
1058         }
1059
1060         pacptr(current_lane)->branch_address = 0;
1061         (void)lm_message("Initiating signature analysis...\n");
1062         (void)lm_message("Hit key to abort.\n");
1063         if(tmg_initiate_play() != SUCCESS)
1064         {
1065             (void)lm_error("Unable to initiate play.\n");
1066             pac_play_cleanup();
1067             return(FAILURE);
1068         }
1069         /* Wait for key hit */
1070         while(lm_check_key() == 0x0)
1071         {
1072             Clear_key_buf();
1073             if(Get_bp_error() == Lane_code(current_lane))
1074             {
1075                 (void)lm_error("Error line asserted on backplane.\n");
1076                 (void)report_bp_error();
1077                 return(FAILURE);
1078             }
1079             else
1080             {
1081                 if(pac_abort_play() != SUCCESS)

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_menu_diag.c	DATE 5/23/89	PAGE # 10/72
			TIME 4:41:20 pm	

LINE #	SOURCE TEXT
1081	{
1082	(void)lm_error("Could not abort play.\n");
1083	return(FAILURE);
1084	}
1085	else
1086	(void)lm_message("Signature analysis halted.\n");
1087	return(SUCCESS);
1088	}
1089	}
1090	}
1091	}
1092	
1093	/*
1094	int pac_patterns_bits_test()
1095	/*
1096	INPUT: none
1097	OUTPUT: return code = SUCCESS or FAILURE
1098	DESCRIPTION:
1099	*/
1100	int
1101	pac_patterns_bits_test()
1102	{
1103	int pam_no;
1104	int pels_found;
1105	int pama_failed;
1106	int pels_failed = 0;
1107	
1108	if(pac_stack_pama(current_lase) != SUCCESS)
1109	{
1110	(void)lm_error("Pattern memory configuration error.\n");
1111	return(FAILURE);
1112	}
1113	
1114	if(diag_clear_errors() != SUCCESS)
1115	{
1116	return(FAILURE);
1117	}
1118	/* See if there are any PELS in the lase */
1119	if((pels_found = pac_check_for_pels()) == 0)
1120	{
1121	(void)lm_error("Can not test pattern bits (no Pin Electronics in lase).\n");
1122	return(FAILURE);
1123	}
1124	
1125	/* Test the PELS one at a time (need to set global PEL variable) */
1126	/* The first PEL which passes the test => the PAC passes and the */
1127	/* test is over */
1128	for(current_pel = 0; current_pel < NUMBER_OF_PELS; ++current_pel)
1129	{
1130	if(((1 << current_pel) & pels_found) == 0) /* current PEL not found */
1131	continue;
1132	
1133	/* Test the pattern bits coming out of each pattern memory on the PAC */
1134	pama_failed = FALSE;
1135	for(pam_no = 0; pam_no < pac(current_lase).sum_pama; ++pam_no)
1136	{
1137	if(pattern_bits_test(pam_no) != SUCCESS)
1138	{
1139	pama_failed = TRUE;
1140	pels_failed  = 1 << current_pel;
1141	if(lm_error("Pattern bits test failed using Pin Electronics id. \n
1142	Pattern Memory id.\n", current_pel, pam_no) != SUCCESS)
1143	return(FAILURE);
1144	}
1145	}
1146	if(pama_failed == FALSE)
1147	{
1148	if(pels_failed == 0)
1149	{
1150	return(SUCCESS);
1151	}
1152	else
1153	{
1154	(void)lm_message("Pattern bits test passed using Pin Electronics id.\n",
1155	current_pel);
1156	}
1157	}
1158	
1159	/* If we got here, then at least one test failed for each PEL */
1160	if((pels_found == 0)    (pama_failed))
1161	{
1162	(void)lm_error("Pattern bits test failed for all Pin Electronics in \n
1163	lase.\n");
1164	}
1165	else
1166	{
1167	(void)lm_message("Pattern bits test passed at least once in lase.\n");
1168	return(SUCCESS);
1169	}
1170	}
1171	
1172	int pac_patterns_bus_test()
1173	/*
1174	INPUT: none
1175	OUTPUT: return code = SUCCESS or FAILURE
1176	DESCRIPTION:
1177	*/
1178	int
1179	pac_patterns_bus_test()
1180	{
1181	int first_pattern;
1182	int pels_found;
1183	int pama_failed;
1184	int pels_failed = 0;
1185	
1186	if(pac_stack_pama(current_lase) != SUCCESS)
1187	{
1188	(void)lm_error("Pattern memory configuration error.\n",
1189	return(FAILURE);
1190	}
1191	
1192	if(diag_clear_errors() != SUCCESS)
1193	{
1194	return(FAILURE);
1195	}
1196	/* See if there are any PELS in the lase */
1197	if((pels_found = pac_check_for_pels()) == 0)
1198	{
1199	(void)lm_error("Cannot test pattern bus (no Pin Electronics in lase).\n");
1200	return(FAILURE);
1201	}
1202	
1203	/* Test the PELS one at a time (need to set global PEL variable) */
1204	/* The first PEL which passes the test => the PAC passes and the */
1205	/* test is over */
1206	for(current_pel = 0; current_pel < NUMBER_OF_PELS; ++current_pel)
1207	{
1208	if(((1 << current_pel) & pels_found) == 0) /* current PEL not found */
1209	continue;
1210	

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_menu\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:20 pm 11/73

```

1201  /* Test all of the pattern memory on the PAC is 128K chunks */
1202  pama_failed = FALSE;
1203  for(first_pattern = 0, first_pattern < pac[current_lane].num_patterns,
1204  first_pattern += PATTERNS_IN_128K)
1205  {
1206      if(patterns_bus_test(first_pattern, PATTERNS_IN_128K) != SUCCESS)
1207      {
1208          pama_failed = TRUE;
1209          pama_failed |= 1 << current_pel;
1210          if(!m_error("Pattern bus test failed using Pin Electronics td, Pattern \
1211 Memory td.\n", current_pel, pac_get_pam_number(first_pattern)) != SUCCESS)
1212              return(FAILURE);
1213      }
1214  }
1215  if(pama_failed == FALSE)
1216  {
1217      if(pama_failed == 0)
1218          return(SUCCESS);
1219      else
1220          (void)m_message("Pattern bus test passed using Pin Electronics td.\n",
1221 current_pel);
1222  }
1223  /* If we got here, then at least one test failed for each PEL */
1224  if((pama_failed == pama_failed))
1225      (void)m_error("Pattern bus test failed for all Pin Electronics in \
1226 lane.\n");
1227  else
1228      (void)m_message("Pattern bus test passed at least once in lane.\n");
1229  return(FAILURE);
1230  }
1231
1232  /*
1233   * int pac_crc_test()
1234   * INPUT: none
1235   * OUTPUT: returns code = SUCCESS or FAILURE
1236   * DESCRIPTION:
1237   */
1238  int
1239  pac_crc_test()
1240  {
1241      int first_pattern;
1242      int dabs_found;
1243      int pama_failed;
1244      int dabs_failed = 0;
1245      if(pac_stack_pama(current_lane) != SUCCESS)
1246      {
1247          (void)m_error("Pattern memory configuration error.\n");
1248          return(FAILURE);
1249      }
1250      if(diag_clear_errors() != SUCCESS)
1251          return(FAILURE);
1252      /* See if there are any Diagnostic Adapters in the lane */
1253      if((dabs_found = pac_check_for_diag_dabs(pac_check_for_pels())) == 0)
1254      {
1255          (void)m_error("Cannot perform CRC (no Diagnostic Adapters in lane).\n");
1256          return(FAILURE);
1257      }
1258      /* Test the PELs one at a time (need to set global PEL variable) */
1259      /* The first PEL which passes the test => the PAC passes and the */
1260      /* test is over */
1261      for(current_pel = 0, current_pel < NUMBER_OF_PELS, ++current_pel)
1262      {
1263          if(((1 << current_pel) & dabs_found) == 0) /* current DAB not found */
1264              continue;
1265          /* Set up the PEL and the DAB for the CRC test (set global dabs_type) */
1266          dabs_type = DIAG_DAB;
1267          if(pel_crc_setup() != SUCCESS)
1268          {
1269              (void)m_error("Unable to perform Pattern Controller CRC test with Pin \
1270 Electronics in slot td.\n", current_pel);
1271              dabs_failed |= 1 << current_pel;
1272              continue;
1273          }
1274          /* Test all of the pattern memory on the PAC is 128K chunks */
1275          pama_failed = FALSE;
1276          for(first_pattern = 0, first_pattern < pac[current_lane].num_patterns,
1277 first_pattern += PATTERNS_IN_128K)
1278          {
1279              if(pel_crc_test_128K(first_pattern) != SUCCESS)
1280              {
1281                  pama_failed = TRUE;
1282                  dabs_failed |= 1 << current_pel;
1283                  if(!m_error("CRC test failed using Pin Electronics td, Pattern \
1284 Memory td.\n", current_pel, pac_get_pam_number(first_pattern)) != SUCCESS)
1285                      return(FAILURE);
1286              }
1287          }
1288          if(pama_failed == FALSE)
1289          {
1290              if(dabs_failed == 0)
1291                  return(SUCCESS);
1292              else
1293                  (void)m_message("CRC test passed using Pin Electronics in slot td.\n",
1294 current_pel);
1295          }
1296      }
1297      /* If we got here, then at least one test failed for each PEL */
1298      if((dabs_found == dabs_failed))
1299          (void)m_error("CRC Test failed for all Pin Electronics in lane.\n");
1300      else
1301          (void)m_message("Pattern bus test passed at least once in lane.\n");
1302      return(FAILURE);
1303  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
1/74

```

LINE # SOURCE TEXT
1  /* SCCS ID: pac_util.c rev 3.1, 4/24/89 at 07:49:27 */
2
3  /*-----*/
4  /* pac_util.c */
5  /*
6  /* Utility routines
7  /* used in PAC diagnostics
8  /*
9  /*-----*/
10 #include <math.h>
11 #include <vrtx.h> /* for lm_delay and lm_time */
12 #include <common.h>
13 #include <mod_def.h>
14 #include <moduler_exta.h>
15 #include <tag.h>
16 #include <tag_def.h>
17 #include <tag_exta.h>
18 #include <tag_run.h>
19 #include <pac.h>
20 #include <magic.h>
21 #include <pel.h>
22 #include <pac_def.h>
23 #include <pac_exta.h>
24 #include <id.h>
25
26
27
28 int
29 pac_check_for_pels()
30 {
31     int slot;
32     int pels_found = 0;
33
34     for(slot = 0; slot < NUMBER_OF_SLOTS; ++slot)
35         pels_found |= (probe_pel(current_lane, slot) == SUCCESS) ? (1 << slot) : 0;
36     return(pels_found);
37 }
38
39 int
40 pac_check_for_diag_dabs(pels_found)
41 int pels_found;
42 {
43     int pel = 0;
44     int diag_dabs = 0;
45
46     pels_found &= ~(0 << NUMBER_OF_PELS);
47     while(pels_found > 0)
48     {
49         if((pels_found & 1) != 0)
50         {
51             if(what_dab(current_lane, pel) == DIAG_DAB)
52                 diag_dabs |= 1 << pel;
53             ++pel;
54             pels_found >>= 1;
55         }
56         return(diag_dabs);
57     }
58 }
59
60 /*
61 /* int pac_pre_play()
62 /*
63 /* INPUT: none
64 /* OUTPUT: returns conservative timeout in ms (0 -> FAILURE)
65 /* DESCRIPTION: Measures clock frequency and then
66 /* sets clock speed register on PAC. Computes and returns
67 /* conservative timeout based on clock period and max
68 /* patterns in enabled lanes.
69 /*
70 int
71 pac_pre_play()
72 {
73     int pac_compute_playtime();
74     int enabled_lanes;
75     int clock_period;
76     int max_patterns;
77     int lane_no;
78
79     /* Find out which lanes have been enabled */
80     enabled_lanes = tsgptr->lane_enable;
81
82     /* Measure the clock period */
83     if((clock_period = tsg_measure_period() / 1000) == 0)
84     {
85         (void)lm_error("Unable to measure the clock period.\n");
86         return(0);
87     }
88
89     /* Set the PAC clock speed registers in each of the enabled and configured
90     /* lanes. Figure out the greatest number of patterns in enabled lanes */
91     for(lane_no = 0; max_patterns = 0; lane_no < NUMBER_OF_LANES; lane_no++)
92     {
93         if((lane_code(lane_no) & enabled_lanes & configured_lanes) != 0)
94         {
95             pac_clock_speed(lane_no, clock_period);
96             if(pac(lane_no).sum_patterns > max_patterns)
97                 max_patterns = pac(lane_no).sum_patterns;
98         }
99     }
100
101     /* Return a timeout based on a worst case expected play time */
102     return(timeout(pac_compute_ time(max_patterns, clock_period)));
103 }
104
105 /*
106 /* int pac_play(timeout)
107 /*
108 /* INPUT: timeout = pattern play timeout in ms
109 /* OUTPUT: returns SUCCESS or FAILURE
110 /* DESCRIPTION: Performs a pattern play.
111 /*
112 int pac_play(timeout)
113 int timeout;
114 {
115     /* Check for errors on the backplane */
116     if((Get_bp_error() & tsgptr->lane_enable) != 0)
117     {
118         report_bp_error();
119         (void)lm_error("Unable to play due to backplane error(s).\n");
120         return(FAILURE);
121     }
122
123     if(tsg_play((long)timeout) != SUCCESS)

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
2/75

```

121  {
122  (void)ls_error("Pattern play fails with a timeout of %dms.\n", timeout);
123  pac_play_cleanup();
124  return(FAILURE);
125  }
126  return(SUCCESS);
127  }
128  }
129  }
130  /*
131  * void pac_info_init()
132  *
133  * INPUT: none
134  * OUTPUT: none
135  * DESCRIPTION: Initializes PAC information table and
136  *             PAC pointers.
137  */
138  void
139  pac_info_init()
140  {
141  int lane_no;
142  for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
143  {
144  pac[lane_no].lane_offset = LANE_A_OFFSET + (lane_no * LANE_SIZE);
145  pac[lane_no].num_patts = 0;
146  pac[lane_no].exists = FALSE;
147  pac[lane_no].num_blocks = Host ? 0x0000 : 01;
148  pac[lane_no].num_patterns = Host ? 0x0000 : 01;
149  pacptr[lane_no] = (PAC *) (pac[lane_no].lane_offset + PAC_RDC_OFFSET);
150  }
151  configured_lanes = NONE;
152  }
153  }
154  /*
155  * int pac_write_patterns(patterns, spatterns_word)
156  *
157  * INPUT: patterns = pattern number (0 counting)
158  *        spatterns_word = address of pattern to be written
159  * OUTPUT: returns SUCCESS or FAILURE
160  * DESCRIPTION: Writes pattern word to pattern location. Returns
161  *             FAILURE if pattern number is outside pattern memory.
162  */
163  int
164  pac_write_patterns(patterns, patptr)
165  int patterns;
166  PAT_WORD *patptr;
167  {
168  u_long location;
169  /* Check if pattern number is valid */
170  if((patterns < 0) || (patterns > (pac[current_lane].num_patterns - 1)))
171  return(FAILURE);
172  /* Compute base address */
173  location = pac[current_lane].lane_offset + (4 * patterns);
174  /* Write pattern word */
175  Write_long(location + BANK_0, patptr->mem_bank[0]);
176  Write_long(location + BANK_1, patptr->mem_bank[1]);
177  Write_long(location + BANK_2, patptr->mem_bank[2]);
178  return(SUCCESS);
179  }
180  }
181  /*
182  * int pac_read_patterns(patterns, spatterns_word)
183  *
184  * INPUT: patterns = pattern number (0 counting)
185  *        spatterns_word = address of pattern to place read values
186  * OUTPUT: returns SUCCESS or FAILURE
187  * DESCRIPTION: Reads pattern word from pattern location. Returns
188  *             FAILURE if pattern number is outside pattern memory.
189  */
190  int
191  pac_read_patterns(patterns, patptr)
192  int patterns;
193  PAT_WORD *patptr;
194  {
195  u_long location;
196  /* Check if pattern number is valid */
197  if((patterns < 0) || (patterns > (pac[current_lane].num_patterns - 1)))
198  return(FAILURE);
199  /* Compute base address */
200  location = pac[current_lane].lane_offset + (4 * patterns);
201  /* Read pattern word */
202  patptr->mem_bank[0] = Read_long(location + BANK_0);
203  patptr->mem_bank[1] = Read_long(location + BANK_1);
204  patptr->mem_bank[2] = Read_long(location + BANK_2);
205  return(SUCCESS);
206  }
207  }
208  /*
209  * int pac_timed_play(clock_period, patterns, play_time)
210  *
211  * INPUT: clock_period = pattern clock period in ns
212  *        patterns = number of patterns to play time before play times out
213  *        play_time = address of play time
214  * OUTPUT: play_time = time of pattern play in ns (5ms resolution)
215  *        returns SUCCESS or FAILURE
216  * DESCRIPTION: Performs pattern play to selected lanes. Both
217  *             the TMC and the PAC must be set up.
218  */
219  int
220  pac_timed_play(clock_period, patterns, play_time)
221  int clock_period;
222  int patterns;
223  int *play_time;
224  {
225  int pac_compute_playtime();
226  int start;
227  int expected_time;
228  int play_result;
229  *play_time = 0;
230  /* Check for errors on the backplane */
231  if((Get_bp_error() & tmcptr->lane_enable) != 0)
232  {
233  report_bp_error();
234  (void)ls_error("Unable to play due to backplane error(s).\n");
235  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

\$

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
3/76

```

LINE # SOURCE TEXT
241     return(FAILURE);
242 }
243
244     expected_time = (long)pac_compute_playtime(patterns, clock_period);
245
246     start = lm_time();           /* Sync time */
247     while(start == lm_time())    /* and */
248     {                           /* then */
249         start = lm_time();       /* Record current time */
250     }
251     play_result = tmg_play((long)Timeout(expected_time));
252     play_time = lm_time() - start; /* Record play time */
253     if(play_result != SUCCESS)
254     {
255         (void)lm_error("tmg_play fails.\n");
256         pac_play_cleanup();
257         return(FAILURE);
258     }
259     /* Check to see if there is an error on the backplane */
260     if(report_bp_error() != 0)
261     {
262         (void)lm_error("Timed pattern play caused backplane error.\n");
263         return(FAILURE);
264     }
265     /* Check to see if actual time was close to expected time */
266     if((play_time < (expected_time - TIMER_RES)) ||
267         (play_time > (expected_time + TIMER_RES)))
268     {
269         (void)lm_error("Play timing error: Expected %d ms. Actual %d ms.\n",
270             expected_time, play_time);
271         return(FAILURE);
272     }
273     return(SUCCESS);
274 }
275
276 void pac_play_cleanup()
277 {
278     INPUT: none
279     OUTPUT: none
280     DESCRIPTION: Performs pattern play cleanup after a failed play.
281 }
282
283 void pac_play_cleanup()
284 {
285     if(bp_mode() == PLAY_MODE)
286     {
287         (void)lm_message("Backplane is still in play mode.\n");
288         (void)lm_message("Attempting to abort play.\n");
289         if(pac_abort_play() != SUCCESS)
290         {
291             (void)lm_message("Abort failed. Forcing backplane into access mode.\n");
292             tmgptr->abort_pattern_play = 1;
293             (void)pac_clock_off();
294             (void)pac_clock_on();
295             lm_delay(5);
296             (void)pac_clock_off();
297             if(bp_mode() == PLAY_MODE)
298             {
299                 (void)lm_message("Unable to force backplane into access mode.\n");
300             }
301         }
302     }
303 }
304
305 void pac_set_first_block(lanes, first_block)
306 {
307     INPUT: lanes = desired lanes for pattern play
308           first_block = block number of 1st pattern
309     OUTPUT: none
310     DESCRIPTION: Sets the branch address register to the
311                desired block number in each of the desired lanes. All
312                desired lanes must be configured.
313 }
314
315 void pac_set_first_block(lanes, first_block)
316 {
317     int lanes;
318     int first_block;
319     for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
320     {
321         if((Lane_code(lane_no) & lanes & configured_lanes) != 0)
322         {
323             pacptr[lane_no]->branch_address = first_block;
324         }
325     }
326 }
327
328 int pac_abort_play()
329 {
330     INPUT: none
331     OUTPUT: returns SUCCESS or FAILURE
332     DESCRIPTION: Aborts pattern play by asserting and then removing
333                the error line.
334 }
335
336 int pac_abort_play()
337 {
338     int start;
339     start = lm_time();
340     tmgptr->backplane_error = 1; /* Assert error line */
341     while(tmgptr->clock_on)      /* Wait for clocks to shut off */
342     {
343         if((lm_time() - start) > 5) /* 5ms timeout */
344         {
345             tmgptr->backplane_error = 0; /* Deassert error line */
346             (void)lm_error("Could not halt play - clocks are on.\n");
347             return(FAILURE);
348         }
349     }
350     tmgptr->backplane_error = 0; /* Deassert error line */
351     /* Flush error signal out of pipeline */
352     if(pac_clock_off() != SUCCESS) /* Turn off clock */
353     {
354         return(FAILURE);
355     }
356     if(pac_clock_on() != SUCCESS) /* Turn on clock */
357     {
358         return(FAILURE);
359     }
360     if(pac_clock_off() != SUCCESS) /* Turn off clock */
361     {
362         return(FAILURE);
363     }
364     if(bp_mode() == PLAY_MODE)
365     {
366         (void)lm_message("Unable to abort play.\n");
367         (void)lm_message("Backplane is still in play mode.\n");
368         return(FAILURE);
369     }
370 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

4/77

```

LINE # SOURCE TEXT
361 }
362 return(SUCCESS);
363 }
364
365
366 /*
367  * int pac_clock_on()
368  * INPUT: none
369  * OUTPUT: returns SUCCESS or FAILURE
370  * DESCRIPTION: Turns on pattern clock.
371  */
372 int
373 pac_clock_on()
374 {
375     int start;
376
377     tmaptr->clock_sync_clear = 1;
378     tmaptr->clock_enable = 1;
379     start = lm_time();
380     while(!tmaptr->clock_on)
381     {
382         if((lm_time() - start) > 5) /* 5ms timeout */
383         {
384             (void)lm_error("Unable to turn on clock.\n");
385             return(FAILURE);
386         }
387     }
388     return(SUCCESS);
389 }
390
391 /*
392  * int pac_clock_off()
393  * INPUT: none
394  * OUTPUT: returns SUCCESS or FAILURE
395  * DESCRIPTION: Turns off pattern clock.
396  */
397 int
398 pac_clock_off()
399 {
400     int start;
401
402     tmaptr->clock_enable = 0;
403     start = lm_time();
404     while(tmaptr->clock_on)
405     {
406         if((lm_time() - start) > 5) /* 5ms timeout */
407         {
408             (void)lm_error("Unable to turn off clock.\n");
409             return(FAILURE);
410         }
411     }
412     tmaptr->clock_sync_clear = 0;
413     return(SUCCESS);
414 }
415
416 /*
417  * int pac_set_pattern_clock(period)
418  * INPUT: period = clock period in nanoseconds
419  * OUTPUT: returns SUCCESS or FAILURE
420  * DESCRIPTION: Sets pattern clock to period specified.
421  * This routine computes the values for n, k, and clock.
422  * select register setting to give the best approximation
423  * of the desired clock period. The clock period provided
424  * is always equal to or greater than that requested.
425  * The actual clock period is related to n, k as follows:
426  * period = (k / n) * T_REF, where
427  * k = 1, 2, ..., 256, n = 2, 4, ..., 512,
428  * n = 128, 129, ..., 256, and
429  * T_REF = 256 / 30 nms = PLL reference period.
430  * Returns: FAILURE if period out of range, else SUCCESS
431  */
432 int
433 pac_set_pattern_clock(period)
434 {
435     double error;
436     double besterror;
437     double kprime;
438     int tmpk;
439     int n;
440     int save_k;
441     int save_n;
442
443     if((period > PAC_MAX_PERIOD) || (period < PAC_MIN_PERIOD))
444         return(FAILURE);
445     if(pac_clock_off() != SUCCESS)
446         return(FAILURE);
447     for(besterror = 1.0, n = N_MIN, k = N_MAX, k++)
448     {
449         kprime = n * period / 8533.3333; /* 8533.3333 = T_REF in ns */
450         if((tmpk = (int)ceil(kprime)) > 512) /* see if out of range */
451             continue; /* if so, ignore it */
452         if(tmpk > 256) /* see if above 256 */
453             continue; /* see if odd */
454         if(tmpk & 1) /* make even */
455             tmpk++;
456         if((error = (((double)tmpk - kprime) / kprime)) < besterror)
457         {
458             save_n = n;
459             save_k = tmpk;
460             besterror = error;
461         }
462     }
463     tmaptr->pll_rate = 256 - save_n + 1; /* put in reg */
464     if(save_k == 1)
465     {
466         tmaptr->pll_divisor = 255; /* special case */
467         tmaptr->clock_select = 0; /* select div by 2 */
468     }
469     else
470     {
471         if(save_k > 256)
472         {
473             tmaptr->pll_divisor = 256 - (save_k >> 1) + 1; /* divide k by 2 */
474             tmaptr->clock_select = 2; /* select div by 4k */
475         }
476         else
477         {
478             tmaptr->pll_divisor = 256 - (save_k >> 1) + 1; /* divide k by 2 */
479             tmaptr->clock_select = 2; /* select div by 4k */
480         }
481     }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE	5/23/89	PAGE #
TIME	4:41:22 pm	5/78

```

LINE # SOURCE TEXT
481 tagptr->pll_divisor = 256 - save_k + 1; /* normal case */
482 tagptr->clock_select = 1; /* select div by 2k */
483 }
484 return(SUCCESS);
485 }
486
487 /*
488  * int pac_stack_pams(lane_no)
489  *
490  * INPUT: lane_no = lane number to stack
491  * OUTPUT: return code = SUCCESS or FAILURE
492  * DESCRIPTION: Reads the PAC COUNT register to see
493  * how many PAMS are stacked onto the PAC. If there
494  * are none, the function returns FAILURE. If there
495  * is at least one PAM, the PAM PRESENT registers
496  * and the PAM ID PROMS are read. If the PAMS are
497  * strapped correctly, the PAC CONFIGURATION register
498  * is written, the PAC INFO structure is filled,
499  * and the function returns SUCCESS. If the PAM
500  * strapping is incorrect the function returns FAILURE.
501  */
502 int
503 pac_stack_pams(lane_no)
504 int lane_no;
505 {
506     if(pac[lane_no].exists != TRUE)
507     {
508         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
509         return(FAILURE);
510     }
511     if(pac_get_sum_pams(lane_no) != SUCCESS)
512     {
513         (void)lm_error("Could not determine number of PAMS in lane %c.\n",
514             'A' + (char)lane_no);
515         return(FAILURE);
516     }
517     if(pam_idprom_test(lane_no) != SUCCESS)
518     {
519         (void)lm_error("Bad ID Prom(s) in PAM(s) in lane %c.\n", 'A' +
520             (char)lane_no);
521         return(FAILURE);
522     }
523     if(pac_get_pam_types(lane_no) != SUCCESS)
524     {
525         (void)lm_error("Could not get PAM types in lane %c.\n", 'A' +
526             (char)lane_no);
527         return(FAILURE);
528     }
529     if(pac_configure_pac(lane_no) != SUCCESS)
530     {
531         (void)lm_error("Could not configure PAC in lane %c.\n", 'A' +
532             (char)lane_no);
533         return(FAILURE);
534     }
535     return(SUCCESS);
536 }
537
538 /*
539  * int pac_get_sum_pams(lane_no)
540  *
541  * INPUT: lane_no = lane number
542  * OUTPUT: return code = SUCCESS or FAILURE
543  * DESCRIPTION: Reads the PAC COUNT register to see
544  * how many PAMS are stacked onto the PAC. If there
545  * are none, the function returns FAILURE. If there
546  * is at least one PAM, the PAM PRESENT registers
547  * are read. If the PAMS are strapped correctly,
548  * the PAC INFO structure is filled, and the function
549  * returns SUCCESS. If the PAM strapping is incorrect
550  * the function returns FAILURE.
551  */
552 int
553 pac_get_sum_pams(lane_no)
554 int lane_no;
555 {
556     int pamno;
557     if(pac[lane_no].exists != TRUE)
558     {
559         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
560         return(FAILURE);
561     }
562     switch(pacptr[lane_no]->pam_count)
563     {
564         case ZERO_PAMS:
565             (void)lm_error("There are no PAMS on the PAC in lane %c.\n",
566                 'A' + (char)lane_no);
567             pac[lane_no].sum_pams = 0;
568             return(FAILURE);
569         case ONE_PAM:
570             pac[lane_no].sum_pams = 1;
571             break;
572         case TWO_PAMS:
573             pac[lane_no].sum_pams = 2;
574             break;
575         case THREE_PAMS:
576             pac[lane_no].sum_pams = 3;
577             break;
578         case FOUR_PAMS:
579             pac[lane_no].sum_pams = 4;
580             break;
581         default:
582             (void)lm_error("PAM Detect register in lane %c returns invalid code.\n",
583                 'A' + (char)lane_no);
584             pac[lane_no].sum_pams = 0;
585             return(FAILURE);
586     }
587     for(pamno = 0; pamno < pac[lane_no].sum_pams; pamno++)
588     {
589         if(pacptr[lane_no]->pam_register[pamno].present == PAM_NOT_PRESENT)
590         {
591             (void)lm_error("PAM %d in lane %c is strapped incorrectly.\n", pamno,
592                 'A' + (char)lane_no);
593             pac[lane_no].sum_pams = 0;
594             return(FAILURE);
595         }
596     }
597 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE	5/23/89	PAGE #
TIME	4:41:22 pm	6/79

```

LINE # SOURCE TEXT
601 }
602 }
603 return(SUCCESS);
604 }
605
606 /*
607  * int pac_get_pam_size(lane_no, pam_no)
608  *
609  * INPUT: lane_no = lane number
610  *        pam_no = Pattern memory board number
611  * OUTPUT: Number of patterns in the board
612  * DESCRIPTION: Reads the PAM ID from to determine the
613  *              number of patterns.
614  */
615 int
616 pac_get_pam_size(lane_no, pam_no)
617 int lane_no;
618 int pam_no;
619 {
620     ID FROM_PAM id_prom_table;
621     ID FROM_PAM *pam_id_info = id_prom_table;
622
623     (void)diag_get_id_info((int)((lane_no * LANE_SIZE) + LANE_A_PAM_ID_PROM +
624     (pam_no * PAM_ID_SPACE)), (char *)pam_id_info);
625
626     return(pam_id_info->patterns << 10);
627 }
628
629 /*
630  * int pac_get_pam_types(lane_no)
631  *
632  * INPUT: lane_no = lane number
633  * OUTPUT: return code = SUCCESS or FAILURE
634  * DESCRIPTION: Checks the PAM ID Proms and if they
635  *              have proper checksums, reads the PAM ID in each.
636  *              If the PAMs are strapped correctly, the PAC INFO
637  *              structure is filled, and the function returns SUCCESS.
638  *              If the PAM strapping is incorrect the function returns
639  *              FAILURE.
640  */
641 int
642 pac_get_pam_types(lane_no)
643 int lane_no;
644 {
645     int pam_no;
646     register u_long current;
647     register u_long patterns;
648
649     pac[lane_no].num_blocks = 0;
650     pac[lane_no].num_patterns = 0;
651
652     if(pac[lane_no].exists != TRUE)
653     {
654         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
655         return(FAILURE);
656     }
657     current = PATTERNS_IN_2M;
658     for(pam_no = 0; pam_no < pac[lane_no].num_pams; pam_no++)
659     {
660         if((patterns = pac_get_pam_size(lane_no, pam_no)) > current)
661         {
662             (void)lm_error("PAMs in lane %c are strapped out of order.\n",
663             'A' + (char)lane_no);
664             pac[lane_no].num_patterns = 0;
665             return(FAILURE);
666         }
667         else
668         {
669             pac[lane_no].p_m_size[pam_no] = current - patterns;
670             pac[lane_no].num_patterns += patterns;
671         }
672     }
673     pac[lane_no].num_blocks = pac[lane_no].num_patterns / BLOCK_SIZE;
674     return(SUCCESS);
675 }
676
677 /*
678  * int pac_configure_pac(lane_no)
679  *
680  * INPUT: lane_no = lane number of PAC to configure
681  * OUTPUT: return code = SUCCESS or FAILURE
682  * DESCRIPTION: Uses values stored in the PAC information
683  *              table (global) to write the PAC CONFIGURATION register.
684  */
685 int
686 pac_configure_pac(lane_no)
687 int lane_no;
688 {
689     if(pac[lane_no].exists != TRUE)
690     {
691         (void)lm_error("PAC in lane %c does not exist.\n", 'A' + (char)lane_no);
692         return(FAILURE);
693     }
694     switch(pac[lane_no].num_pams)
695     {
696         case 1:
697             switch(pac[lane_no].num_blocks)
698             {
699                 case BLKS_IN_128K:
700                     pacptr[lane_no]->configuration = 60;
701                     break;
702                 case BLKS_IN_512K:
703                     pacptr[lane_no]->configuration = 61;
704                     break;
705                 case BLKS_IN_2M:
706                     pacptr[lane_no]->configuration = 62;
707                     break;
708                 default:
709                     (void)lm_error("Number of blocks in information table invalid.\n");
710                     break;
711             }
712         case 2:
713         case 3:
714         case 4:
715             pacptr[lane_no]->configuration = (pac[lane_no].num_blocks /
716             BLKS_IN_128K) - 1;
717             break;
718         default:
719             (void)lm_error("Number of PAMs in information table invalid.\n");
720             return(FAILURE);
721     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

7/80

LINE # SOURCE TEXT

```

721 Pac_set_parity(lane_no, SET_ODD_PARITY);
722 return(SUCCESS);
723 }
724
725 /*
726  * int pac_fill_check(first_pattern, patterns)
727  * INPUT: first_pattern = first pattern number to fill (0 counting)
728  * patterns = number of patterns to fill
729  * OUTPUT: return code = SUCCESS or FAILURE
730  * DESCRIPTION: Checks bounds on first_pattern and patterns.
731  * Used by pac_fill_xx functions. The function returns
732  * SUCCESS if the patterns to be filled
733  * are within the pattern memory and FAILURE if any
734  * portion of the fill space is outside the physical
735  * pattern memory address space.
736  */
737
738 int
739 pac_fill_check(first_pattern, patterns)
740 int first_pattern;
741 int patterns;
742 {
743     if((first_pattern < 0) ||
744         (first_pattern > (pac[current_lane].num_patterns - 1)))
745     {
746         (void)lm_error("First pattern number passed to fill routine \
747 outside of pattern memory.\n");
748         return(FAILURE);
749     }
750     if((patterns < 1) || ((first_pattern + patterns) >
751         pac[current_lane].num_patterns))
752     {
753         (void)lm_error("Number of patterns passed to fill routine \
754 outside of pattern memory.\n");
755         return(FAILURE);
756     }
757     return(SUCCESS);
758 }
759
760 /*
761  * int pac_fill_random(first_pattern, patterns, fill_mask, seed)
762  * INPUT: first_pattern = first pattern number to fill
763  * patterns = number of patterns to fill
764  * fill_mask = control word mask
765  * seed = random number seed
766  * OUTPUT: return code = SUCCESS or FAILURE
767  * DESCRIPTION: Fills pattern memory with pseudorandom
768  * data starting at the pattern number specified by 'first_pattern'.
769  * The total number of patterns is specified by 'patterns'.
770  * The function returns SUCCESS if the operation is
771  * successful and FAILURE if any portion of the fill
772  * space is outside the physical pattern memory address
773  * space.
774  */
775
776 int
777 pac_fill_random(first_pattern, patterns, fill_mask, seed)
778 int first_pattern;
779 int patterns;
780 register u_long fill_mask;
781 u_long seed;
782 {
783     register u_long *memptr;
784     register u_long i;
785     register u_long random_number;
786     u_long pattern_offset;
787     u_long bank_offset;
788     int bank;
789
790     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
791         return(FAILURE);
792
793     pattern_offset = pac[current_lane].lane_offset + 4*first_pattern;
794     random_number = seed;
795
796     (void)lm_message("Filling to patterns with random data", patterns);
797     for(bank = 0; bank_offset = 0; bank < 3; ++bank)
798     {
799         (void)lm_message(" ");
800         memptr = (u_long *) (pattern_offset + bank_offset);
801         i = patterns / 4;
802         if((bank == 2) && (fill_mask != "01"))
803         {
804             do
805             {
806                 random_number = Pac_get_random(random_number);
807                 *memptr++ = random_number & fill_mask;
808                 random_number = Pac_get_random(random_number);
809                 *memptr++ = random_number & fill_mask;
810                 random_number = Pac_get_random(random_number);
811                 *memptr++ = random_number & fill_mask;
812                 random_number = Pac_get_random(random_number);
813                 *memptr++ = random_number & fill_mask;
814             } while (--i);
815             for(i=0; i < (patterns % 4); ++i)
816             {
817                 random_number = Pac_get_random(random_number);
818                 *memptr++ = random_number & fill_mask;
819             }
820         }
821         else
822         {
823             do
824             {
825                 random_number = Pac_get_random(random_number);
826                 *memptr++ = random_number;
827                 random_number = Pac_get_random(random_number);
828                 *memptr++ = random_number;
829                 random_number = Pac_get_random(random_number);
830                 *memptr++ = random_number;
831                 random_number = Pac_get_random(random_number);
832                 *memptr++ = random_number;
833             } while (--i);
834             for(i=0; i < (patterns % 4); ++i)
835             {
836                 random_number = Pac_get_random(random_number);
837                 *memptr++ = random_number;
838             }
839         }
840         bank_offset += PAT_MEM_BANK;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

8/81

```

LINE # SOURCE TEXT
841 }
842 (void)lm_message("done.\n");
843 return(SUCCESS);
844 }
845
846 /*
847  * int pac_read_random(first_pattern, patterns, fill_mask, seed)
848  *
849  * INPUT:  first_pattern = first pattern number to read
850  *         patterns = number of patterns to read
851  *         fill_mask = control word mask
852  *         seed = random number seed
853  *
854  * OUTPUT: return code = SUCCESS or FAILURE
855  *
856  * DESCRIPTION: Reads pattern memory and compares with pseudorandom
857  * data starting at the pattern number specified by 'first_pattern'.
858  * The total number of patterns is specified by 'patterns'.
859  * The function returns SUCCESS if all patterns are identical.
860  * The function returns FAILURE if any of the patterns do not
861  * compare. The failed address and the good and bad data words
862  * are output as an error message.
863  */
864 int
865 pac_read_random(first_pattern, patterns, fill_mask, seed)
866 {
867     register int patterns;
868     register u_long fill_mask;
869     register u_long seed;
870
871     register u_long *memptr;
872     register u_long i;
873     register u_long temp;
874     register u_long random_number;
875     u_long pattern_offset;
876     u_long bank_offset;
877     int bank;
878     int returncode = SUCCESS;
879
880     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
881         return(FAILURE);
882     pattern_offset = pac(current_lane).lane_offset + 4 * first_pattern;
883     random_number = seed;
884
885     (void)lm_message("Reading rd random data patterns", patterns);
886     for(bank = 0, bank_offset = 0; bank < 3; ++bank)
887     {
888         (void)lm_message("-");
889         memptr = (u_long *) (pattern_offset + bank_offset);
890         if((bank == 2) && (fill_mask != '01'))
891         {
892             for(i=0; i < patterns; ++i)
893             {
894                 random_number = Pac_get_random(random_number);
895                 if((temp = *memptr++) != (random_number & fill_mask))
896                 {
897                     if(lm_error("Random read test. Address = %08x.\n",
898                         Expected = %08x, Actual = %08x.\n", memptr - 1, random_number & fill_mask,
899                         temp) != SUCCESS)
900                         return(FAILURE);
901                     else
902                         returncode = FAILURE;
903                 }
904             }
905         }
906         else
907         {
908             for(i=0; i < patterns; ++i)
909             {
910                 random_number = Pac_get_random(random_number);
911                 if((temp = *memptr++) != random_number)
912                 {
913                     if(lm_error("Random read test. Address = %08x.\n",
914                         Expected = %08x, Actual = %08x.\n", memptr - 1, random_number, temp) != SUCCESS)
915                         return(FAILURE);
916                     else
917                         returncode = FAILURE;
918                 }
919             }
920         }
921         bank_offset += PAT_MEM_BANK;
922     }
923     (void)lm_message("done.\n");
924     return(returncode);
925 }
926
927 /*
928  * int pac_fill_counting(first_pattern, patterns, fill_mask)
929  *
930  * INPUT:  first_pattern = first pattern
931  *         patterns = number of patterns to fill
932  *         fill_mask = control word mask
933  *
934  * OUTPUT: return code = SUCCESS or FAILURE
935  *
936  * DESCRIPTION: Fills pattern memory with 'counting' data
937  * starting at the pattern number specified by 'first_pattern'.
938  * The total number of patterns is specified by 'patterns'.
939  * The function returns SUCCESS if the operation is successful
940  * and FAILURE if any portion of the fill space is outside
941  * the physical pattern memory address space.
942  */
943 int
944 pac_fill_counting(first_pattern, patterns, fill_mask)
945 {
946     register int patterns;
947     register u_long fill_mask;
948
949     register u_long i;
950     register u_long location;
951
952     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
953         return(FAILURE);
954     location = pac(current_lane).lane_offset + 4 * first_pattern;
955     (void)lm_message("Filling rd patterns with counting data...", patterns);
956     for(i=0; i < patterns; ++i)
957     {
958         /* Write count to Banks 0 and 1 */
959         Write_long(location + BANK_0, i);
960         Write_long(location + BANK_1, i);
961         /* Write count to Bank 2. Masking off control bits */
962         Write_long(location + BANK_2, i & fill_mask);
963         location += 4; /* Increment location to next long word address */
964     }
965     (void)lm_message("done.\n");
966 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

9/82

```

LINE # SOURCE TEXT
961 return(SUCCESS);
962 }
963
964
965 /* int pac_fill_load0(first_pattern, patterns, fill_mask)
966 *
967 * INPUT: first_pattern = first pattern
968 * patterns = number of patterns to fill
969 * fill_mask = control word mask
970 * OUTPUT: return code = SUCCESS or FAILURE
971 * DESCRIPTION: Fills pattern memory with alternating ones
972 * and zeros data starting at the pattern number specified by
973 * first_pattern. The total number of patterns is specified by
974 * patterns. The function returns SUCCESS if the operation
975 * is successful and FAILURE if any portion of the fill
976 * space is outside the physical pattern memory address space.
977 */
978
979 int
980 pac_fill_load0(first_pattern, patterns, fill_mask)
981 int first_pattern;
982 register int patterns;
983 register u_long fill_mask;
984 {
985     register u_long i;
986     register u_long value;
987     register u_long location;
988
989     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
990         return(FAILURE);
991     location = pac[current_lane].lane_offset + 4*first_pattern;
992     (void)lm_message("Filling 0d patterns with alternating ones and \
993     zeros data...", patterns);
994     for(i=0; i < patterns; i++)
995     {
996         if((i & 1) == 0) /* If pattern number is even */
997             value = 0; /* value is all zeros */
998         else
999             value = ~0; /* value is all ones */
1000         /* Write value to Banks 0 and 1 */
1001         Write_long(location + BANK_0, value);
1002         Write_long(location + BANK_1, value);
1003         /* Write zeros to BANK 2, masking off control bits */
1004         Write_long(location + BANK_2, value & fill_mask);
1005         location += 4; /* Increment location to next long word address */
1006     }
1007     (void)lm_message("Done.\n");
1008     return(SUCCESS);
1009 }
1010
1011 /* int pac_fill_walking(first_pattern, patterns, fill_mask, data)
1012 *
1013 * INPUT: first_pattern = block address of first pattern
1014 * patterns = number of patterns to fill
1015 * fill_mask = control word mask
1016 * data = pattern data (0 or 1)
1017 * OUTPUT: return code = SUCCESS or FAILURE
1018 * DESCRIPTION: Fills pattern memory with walking ones or
1019 * zeros data starting at the block number specified by
1020 * first_pattern. The total number of patterns is specified by
1021 * patterns. The function returns SUCCESS if the operation
1022 * is successful and FAILURE if any portion of the fill
1023 * space is outside the physical pattern memory address space.
1024 */
1025
1026 int
1027 pac_fill_walking(first_pattern, patterns, fill_mask, data)
1028 int first_pattern;
1029 register int patterns;
1030 u_long fill_mask;
1031 int data;
1032 {
1033     register u_long i;
1034     u_long same;
1035     u_long shift[32];
1036     register u_long location;
1037
1038     if(pac_fill_check(first_pattern, patterns) != SUCCESS)
1039         return(FAILURE);
1040     if(data == 0) /* Fill array with walking zeros */
1041     {
1042         same = 0;
1043         shift[0] = ~1;
1044         for(i=1; i < 32; ++i)
1045             shift[i] = (shift[i-1] << 1) | 0x01;
1046     }
1047     else /* Fill array with walking ones */
1048     {
1049         same = 0;
1050         shift[0] = 1;
1051         for(i=1; i < 32; ++i)
1052             shift[i] = (shift[i-1] << 1);
1053     }
1054     location = pac[current_lane].lane_offset + 4*first_pattern;
1055     (void)lm_message("Filling 0d patterns with walking %s data...",
1056     patterns, (data == 0) ? "zeros" : "ones");
1057     for(i=0; i < patterns; i++)
1058     {
1059         switch((i/32)%3)
1060         {
1061             case 0:
1062                 Write_long(location + BANK_0, shift[(i/32)]);
1063                 Write_long(location + BANK_1, same);
1064                 Write_long(location + BANK_2, same & fill_mask);
1065                 break;
1066             case 1:
1067                 Write_long(location + BANK_0, same);
1068                 Write_long(location + BANK_1, shift[(i/32)]);
1069                 Write_long(location + BANK_2, same & fill_mask);
1070                 break;
1071             case 2:
1072                 Write_long(location + BANK_0, same);
1073                 Write_long(location + BANK_1, same);
1074                 Write_long(location + BANK_2, shift[(i/32) & fill_mask]);
1075                 break;
1076             default:
1077                 return(FAILURE);
1078                 break;
1079         }
1080         location += 4; /* Increment location to next long word address */
1081     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

10/83

```

1081 (void)lm_message("done.\n");
1082 return(SUCCESS);
1083 }
1084
1085 /*
1086  * int location_test(address, first_bit, num_bits)
1087  *
1088  * INPUT: address = long word address
1089  *        first_bit = first valid bit in register (0-31)
1090  *        num_bits = number of valid bits in register (1-32)
1091  * OUTPUT: returns SUCCESS or FAILURE
1092  * DESCRIPTION: Performs complete register (or memory location)
1093  * test on location pointed to by address. The first and last
1094  * bits in the register determine which bits the function checks.
1095  */
1096 int
1097 location_test(address, first_bit, num_bits)
1098 u_long address;
1099 int first_bit;
1100 int num_bits;
1101 {
1102     u_long i;
1103     int j;
1104     u_long mask;
1105     u_long temp;
1106     int returncode = SUCCESS;
1107
1108     mask = (~0L << first_bit) & ~(~0L << (first_bit + num_bits));
1109
1110     /* Perform walking ones test */
1111     for(i = 1 << first_bit; j = 0; j < num_bits; j++, i <<= 1)
1112     {
1113         Write_long(address, i); /* Write pattern to location */
1114         if((temp = (Read_long(address) & mask)) != i)
1115         {
1116             returncode = FAILURE;
1117             if(lm_error("PAC location test. Address = %08x.\n",
1118 \Expected = %08x.\n, Actual = %08x.\n", address, i, temp) != SUCCESS)
1119                 return(FAILURE);
1120         }
1121     }
1122
1123     /* Perform walking zeros test */
1124     for(i = (~0L << first_bit) & mask; j = 0; j < num_bits; j++,
1125 i = ((i << 1) | 0x01) & mask)
1126     {
1127         Write_long(address, i); /* Write pattern to location */
1128         if((temp = (Read_long(address) & mask)) != i)
1129         {
1130             returncode = FAILURE;
1131             if(lm_error("PAC location test. Address = %08x.\n",
1132 \Expected = %08x.\n, Actual = %08x.\n", address, i, temp) != SUCCESS)
1133                 return(FAILURE);
1134         }
1135     }
1136     return(returncode);
1137 }
1138
1139 /*
1140  * int build_pattern_control(first_pattern, patterns, mode)
1141  *
1142  * INPUT: first_pattern = first pattern number to link
1143  *        patterns = number of patterns to link
1144  *        mode = STOP MODE or LOOP MODE
1145  * OUTPUT: return code = SUCCESS or FAILURE
1146  * DESCRIPTION: Links blocks of pattern memory with
1147  * link table. Puts Branch Always instructions in the
1148  * proper locations. In STOP MODE, a stop instruction is
1149  * placed at the end of the pattern sequence. In LOOP MODE,
1150  * a Branch Always instruction is placed at the end of the
1151  * pattern sequence and the link table is loaded so that
1152  * the last branch is to the initial pattern block. The
1153  * function returns SUCCESS or FAILURE.
1154  */
1155 int
1156 build_pattern_control(first_pattern, patterns, mode)
1157 int first_pattern;
1158 register int patterns;
1159 int mode;
1160 {
1161     register int i;
1162     register int j;
1163     register int max_index;
1164     register int total_branches;
1165     register u_long *memptr;
1166     u_long *linkptr;
1167     u_long pat_num;
1168     u_long link_table;
1169     int first_block;
1170
1171     if((first_pattern < 0) ||
1172 (first_pattern > pac(current_lane).num_patterns - 1))
1173     {
1174         (void)lm_error("First pattern number passed to fill routine \
1175 outside of pattern memory.\n");
1176         return(FAILURE);
1177     }
1178     if((patterns < 1) || (patterns > pac(current_lane).num_patterns))
1179     {
1180         (void)lm_error("Number of patterns passed to fill routine \
1181 outside of pattern memory.\n");
1182         return(FAILURE);
1183     }
1184     if((first_pattern & BLOCK_SIZE) != 0)
1185     {
1186         (void)lm_error("First pattern in seq not on block boundary.\n");
1187         return(FAILURE);
1188     }
1189     if((mode == LOOP_MODE) && ((patterns & BLOCK_SIZE) != 0))
1190     {
1191         (void)lm_error("Can only loop in block increments.\n");
1192         return(FAILURE);
1193     }
1194
1195     /* Compute first block number and total number of branches */
1196     first_block = first_pattern/BLOCK_SIZE;
1197     pacptr(current_lane)->branch_address = first_block;
1198     total_branches = (patterns - 1)/BLOCK_SIZE;
1199
1200     /* Compute first address of pattern memory and link table */

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pac\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:22 pm

11/84

```

1201 pat_mem = pac(current_lane).lane_offset + BANK_2;
1202 memptr = (u_long *) (pat_mem + 4 * first_patterns);
1203 link_table = pac(current_lane).lane_offset + LINK_OFFSET;
1204 linkptr = (u_long *) (link_table + 4 * first_block);
1205 /* See if patterns sequence wraps around pattern memory */
1206 if ((first_patterns + patterns) <= pac(current_lane).num_patterns) /* No */
1207 {
1208     /* Put NOP instructions (Don't branch or stop) in pattern sequence */
1209     for (i=0; i < patterns; i++)
1210     {
1211         *(memptr++) = NOP_MASK;
1212     }
1213
1214     /* Fill link table and place branch always instructions */
1215     memptr = (u_long *) (pat_mem + 4 * (first_patterns + BLOCK_SIZE -
1216     BRANCH_LATENCY));
1217     for (i = first_block + 1, j = 0; j < total_branches; i++, j++)
1218     {
1219         *(linkptr++) = i; /* Connect blocks in link table */
1220         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1221         memptr += BLOCK_SIZE; /* Go to next branch location */
1222     }
1223
1224     /* patterns sequence wraps around to beginning of pattern memory */
1225     /* Put NOP instructions (Don't branch or stop) in pattern sequence */
1226     max_index = pac(current_lane).num_patterns - first_patterns;
1227     for (i=0; i < max_index; i++)
1228     {
1229         *(memptr++) = NOP_MASK;
1230     }
1231     memptr = (u_long *) (pat_mem);
1232     max_index = patterns - max_index;
1233     for (i=0; i < max_index; i++)
1234     {
1235         *(memptr++) = NOP_MASK;
1236     }
1237
1238     /* Fill link table and place branch always instructions */
1239     memptr = (u_long *) (pat_mem + 4 * (first_patterns + BLOCK_SIZE -
1240     BRANCH_LATENCY));
1241     max_index = pac(current_lane).num_blocks - first_block;
1242     for (i = first_block + 1, j = 0; j < max_index; i++, j++)
1243     {
1244         *(linkptr++) = i; /* Connect blocks in link table */
1245         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1246         memptr += BLOCK_SIZE; /* Go to next branch location */
1247     }
1248     *(--linkptr) = 0; /* Point back to first block */
1249     linkptr = (u_long *) (link_table);
1250     memptr = (u_long *) (pat_mem + 4 * (BLOCK_SIZE - BRANCH_LATENCY));
1251     max_index = total_branches - max_index;
1252     for (i = 0; i < max_index; i++)
1253     {
1254         *(linkptr++) = i + 1; /* Connect blocks in link table */
1255         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1256         memptr += BLOCK_SIZE; /* Go to next branch location */
1257     }
1258
1259     linkptr = first_block; /* Go back to first block */
1260
1261     /* Place stop or branch instruction, depending on mode */
1262     if (mode == LOOP_MODE)
1263     {
1264         *memptr |= BRANCH_ALWAYS; /* Place branch always instruction */
1265     }
1266     else /* Must be STOP_MODE */
1267     {
1268         memptr = (u_long *) (pat_mem + 4 * ((first_patterns + patterns -
1269     STOP_LATENCY) & pac(current_lane).num_patterns));
1270         *memptr |= STOP; /* Place stop instruction */
1271     }
1272     return(SUCCESS);
1273 }
1274
1275 void pac_clear_link()
1276 {
1277     /*
1278     * INPUT: none
1279     * OUTPUT: none
1280     * DESCRIPTION: Clears link table in current lane.
1281     */
1282 }
1283
1284 void pac_clear_pam()
1285 {
1286     register u_long *memptr;
1287     register u_long i;
1288
1289     memptr = (u_long *) (pac(current_lane).lane_offset + LINK_OFFSET);
1290
1291     i = (LINK_SIZE >> 2) / 8;
1292     do
1293     {
1294         *memptr++ = 0;
1295         *memptr++ = 0;
1296         *memptr++ = 0;
1297         *memptr++ = 0;
1298         *memptr++ = 0;
1299         *memptr++ = 0;
1300         *memptr++ = 0;
1301         *memptr++ = 0;
1302     } while(--i);
1303 }
1304
1305 int pac_clear_pam(lane_no, pam_no)
1306 {
1307     /*
1308     * INPUT: lane_no = lane number of PAC/PAMs
1309     *        pam_no = PAM number
1310     * OUTPUT: return code = SUCCESS or FAILURE
1311     * DESCRIPTION: Clears pattern memory on specified PAM.
1312     */
1313     int
1314     pac_clear_pam(lane_no, pam_no)
1315     int lane_no;
1316     int pam_no;
1317     {
1318         register u_long *memptr;
1319         register u_long i;
1320         register u_long total_patterns;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
12/85

```

1321 int bank,
1322 u_long bank_offset,
1323
1324 if(pac(lane_no).exists != TRUE)
1325 {
1326     (void)lm_error("PAC in lane %d does not exist.\n", 'A' + (char)lane_no);
1327     return(FAILURE);
1328 }
1329 if ((total_patterns = pac(lane_no).pam_size(pam_no)) == 0)
1330 {
1331     return SUCCESS; /* nothing to clear */
1332 }
1333 (void)lm_message("Clearing PAM %d (bank) in lane %d", pam_no,
1334 total_patterns >> 10, 'A' + (char)lane_no);
1335 for(bank = 0, bank_offset = 0; bank < 3; ++bank)
1336 {
1337     lm_message("."); /* walking period before each bank */
1338     samptr = (u_long)(Pam_base_address(lane_no, pam_no) + bank_offset);
1339     i = total_patterns / 8;
1340     do
1341     {
1342         *samptr++ = 0;
1343         *samptr++ = 0;
1344         *samptr++ = 0;
1345         *samptr++ = 0;
1346         *samptr++ = 0;
1347         *samptr++ = 0;
1348         *samptr++ = 0;
1349         *samptr++ = 0;
1350     } while(--i);
1351     bank_offset += PAT_BANK_SIZE;
1352 }
1353 (void)lm_message("Done.\n");
1354 return(SUCCESS);
1355 }
1356
1357 /*
1358 int pac_clear_pam_mem(lane_no)
1359 *
1360 * INPUT: lane_no = lane number of PAC to clear
1361 * OUTPUT: return code = SUCCESS or FAILURE
1362 * DESCRIPTION: Clears all of pattern memory.
1363 */
1364 int
1365 pac_clear_pam_mem(lane_no)
1366 int lane_no,
1367 {
1368     int pam_no;
1369     for(pam_no = 0; pam_no < pac(lane_no).num_pams; pam_no++)
1370     {
1371         if(pac_clear_pam(lane_no, pam_no) != SUCCESS)
1372         {
1373             (void)lm_error("Unable to clear pattern memory in lane %d.\n",
1374 'A' + (char)lane_no);
1375             return(FAILURE);
1376         }
1377     }
1378     return(SUCCESS);
1379 }
1380
1381 /*
1382 int pac_compute_playtime(count, clock_period)
1383 *
1384 * INPUT: count = length of pattern play in physical clocks
1385 *        clock_period = clock period in ns
1386 * OUTPUT: play time in ns
1387 * DESCRIPTION: Computes duration of pattern play in milliseconds.
1388 */
1389 int
1390 pac_compute_playtime(count, clock_period)
1391 int count,
1392 int clock_period,
1393 {
1394     return(((int)((double)count * (double)clock_period) /
1395 1000000.0));
1396 }
1397
1398 #define MIN_SAMPLE_PULSE 500 /* Min sample pulse width = 500 ns */
1399
1400 /*
1401 int pac_compute_sample_width(clock_period)
1402 *
1403 * INPUT: clock_period = clock period in ns
1404 * OUTPUT: sample width in clock cycles
1405 * DESCRIPTION: Computes sample width for pattern play.
1406 */
1407 int
1408 pac_compute_sample_width(clock_period)
1409 int clock_period,
1410 {
1411     int sample_width;
1412     sample_width = (u_long)MIN_SAMPLE_PULSE / clock_period;
1413     return((sample_width < 1) ? 1 : sample_width);
1414 }
1415
1416 /*
1417 void pac_probe_all_pacs()
1418 *
1419 * INPUT: none
1420 * OUTPUT: none
1421 * DESCRIPTION: Probes all PACs in the modeler and
1422 *             fills in the .exists PAC information attribute.
1423 */
1424 void
1425 pac_probe_all_pacs()
1426 {
1427     register int lane;
1428     for(lane = 0; lane < NUMBER_OF_LANES; lane++)
1429     {
1430         pac(lane).exists = (probe_pac(lane) == SUCCESS)? TRUE : FALSE;
1431     }
1432 }
1433
1434 /*
1435 void pac_configure_all_pacs()
1436 *
1437 * INPUT: none
1438 * OUTPUT: none
1439 * DESCRIPTION: Configures all PACs in the modeler.
1440 */

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
13/86

```

1441 void
1442 pac_configure_all_pacs()
1443 {
1444     int lane_no;
1445
1446     pac_info_init(); /* Initialize PAC information structure */
1447     pac_probe_all_pacs();
1448     for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)
1449     {
1450         if (pac(lane_no).exists == TRUE)
1451         {
1452             if(pac_stack_pacs(lane_no) == SUCCESS)
1453             {
1454                 if(pac_clear_pat_mem(lane_no) == SUCCESS)
1455                 {
1456                     configured_lanes |= Lane_code(lane_no);
1457                 }
1458             }
1459         }
1460     }
1461
1462     /*
1463     * int pac_insert_parity_error(pattern, word_no)
1464     *
1465     * INPUT: pattern = pattern number to insert parity error
1466     *        word_no = word number within pattern number
1467     * OUTPUT: return code = SUCCESS or FAILURE
1468     * DESCRIPTION: Inserts a parity error in pattern memory
1469     *              at the location specified by the pattern number and the
1470     *              word number. The word number is encoded as follows
1471     *              0 = control word
1472     *              1 = pattern bits 0-15
1473     *              2 = pattern bits 16-31
1474     *              3 = pattern bits 32-47
1475     *              4 = pattern bits 48-63
1476     *              5 = pattern bits 64-79
1477     */
1478     int
1479     pac_insert_parity_error(pattern, word_no)
1480     int pattern;
1481     int word_no;
1482     {
1483         u_long bank;
1484         u_long word;
1485         u_long address;
1486         u_long contents;
1487
1488         /* Check bounds on pattern */
1489         if((pattern < 0) || (pattern > (pac(current_lane).num_patterns - 1)))
1490         {
1491             (void)fprintf(stderr, "Pattern number out of bounds.\n");
1492             return(FAILURE);
1493         }
1494         /* decode word number */
1495         switch(word_no)
1496         {
1497             case 0:
1498                 bank = BANK_2;
1499                 word = 2;
1500                 break;
1501             case 1:
1502                 bank = BANK_2;
1503                 word = 0;
1504                 break;
1505             case 2:
1506                 bank = BANK_1;
1507                 word = 2;
1508                 break;
1509             case 3:
1510                 bank = BANK_1;
1511                 word = 0;
1512                 break;
1513             case 4:
1514                 bank = BANK_0;
1515                 word = 2;
1516                 break;
1517             case 5:
1518                 bank = BANK_0;
1519                 word = 0;
1520                 break;
1521             default:
1522                 (void)fprintf(stderr, "Word number out of bounds.\n");
1523                 return(FAILURE);
1524                 break;
1525         }
1526         /* Compute address */
1527         address = pac(current_lane).lane_offset + bank + (4 * pattern) + word;
1528         contents = Read_word(address);
1529
1530         /* change parity to opposite of current state */
1531         if(word == 0)
1532             pacptr[current_lane]-->high_word_parity =
1533             !pacptr[current_lane]-->high_word_parity;
1534         else
1535             pacptr[current_lane]-->low_word_parity =
1536             !pacptr[current_lane]-->low_word_parity;
1537         Write_word(address, contents);
1538
1539         /* change parity back to original state */
1540         if(word == 0)
1541             pacptr[current_lane]-->high_word_parity =
1542             !pacptr[current_lane]-->high_word_parity;
1543         else
1544             pacptr[current_lane]-->low_word_parity =
1545             !pacptr[current_lane]-->low_word_parity;
1546         return(SUCCESS);
1547     }
1548
1549     /*
1550     * int pac_remove_parity_error(pattern, word_no)
1551     *
1552     * INPUT: pattern = pattern number with parity error
1553     *        word_no = word number within pattern number
1554     * OUTPUT: return code = SUCCESS or FAILURE
1555     * DESCRIPTION: Removes a parity error in pattern memory
1556     *              at the location specified by the pattern number and the
1557     *              word number. The word number is encoded as follows
1558     *              0 = control word
1559     *              1 = pattern bits 0-15
1560     */

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pac_util.c	DATE 5/23/89	PAGE # 14/87
LINE #		SOURCE TEXT		
1561	*	2 = patterns bits 16-31		
1562	*	3 = patterns bits 32-47		
1563	*	4 = patterns bits 48-63		
1564	*	5 = patterns bits 64-79		
1565	*/			
1566	int			
1567	pac_remove_parity_error(patterns, word_no)			
1568	int patterns;			
1569	int word_no;			
1570	{			
1571	u_long bank;			
1572	u_long word;			
1573	u_long address;			
1574	u_long contents;			
1575	u_long contents;			
1576	/* Check bounds on patterns */			
1577	if((patterns < 0)    (patterns > (pac[current_lane].num_patterns - 1)))			
1578	{			
1579	(void)lm_error("Pattern number out of bounds.\n");			
1580	return(FAILURE);			
1581	}			
1582	/* decode word number */			
1583	switch(word_no)			
1584	{			
1585	case 0:			
1586	bank = BANK_2;			
1587	word = 2;			
1588	break;			
1589	case 1:			
1590	bank = BANK_2;			
1591	word = 0;			
1592	break;			
1593	case 2:			
1594	bank = BANK_1;			
1595	word = 2;			
1596	break;			
1597	case 3:			
1598	bank = BANK_1;			
1599	word = 0;			
1600	break;			
1601	case 4:			
1602	bank = BANK_0;			
1603	word = 2;			
1604	break;			
1605	case 5:			
1606	bank = BANK_0;			
1607	word = 0;			
1608	break;			
1609	default:			
1610	(void)lm_error("Word number out of bounds.\n");			
1611	return(FAILURE);			
1612	break;			
1613	}			
1614	/* Compute address */			
1615	address = pac[current_lane].lane_offset + bank + (4 * patterns) + word;			
1616	contents = Read_word(address);			
1617	Write_word(address, contents);			
1618	Clear_pac_errors(current_lane);			
1619	return(SUCCESS);			
1620	}			
1621	}			
1622	/*			
1623	int pac_get_pam_number(patterns_no)			
1624	/*			
1625	INPUT: patterns_no = pattern number			
1626	OUTPUT: Returns board number containing pattern number			
1627	DESCRIPTION: Determines which Pattern Memory board			
1628	contains the specified pattern number. The function			
1629	returns -1 if the pattern number is outside of pattern			
1630	memory.			
1631	*/			
1632	int			
1633	pac_get_pam_number(patterns_no)			
1634	register int patterns_no;			
1635	{			
1636	register int total_patterns;			
1637	register int pam_no;			
1638	register int total_pams = pac[current_lane].num_pams;			
1639	if((patterns_no < 0)    (patterns_no > (pac[current_lane].num_patterns - 1)))			
1640	return(-1);			
1641	for(total_patterns = 0; pam_no = 0; pam_no < total_pams; ++pam_no)			
1642	{			
1643	total_patterns += pac[current_lane].pam_size[pam_no];			
1644	if(patterns_no < total_patterns)			
1645	return(pam_no);			
1646	}			
1647	return(-1);			
1648	}			
1649	/*			
1650	int pac_get_first_pattern_no(lane_no, pam_no)			
1651	/*			
1652	INPUT: lane_no = lane number			
1653	pam_no = pattern memory board number			
1654	OUTPUT: Returns first pattern number in specified pattern memory board			
1655	DESCRIPTION: Determines the first pattern number (zeros counting)			
1656	in the specified Pattern Memory board.			
1657	*/			
1658	int			
1659	pac_get_first_pattern_no(lane_no, pam_no)			
1660	register int lane_no;			
1661	register int pam_no;			
1662	{			
1663	register int first_pattern = 0;			
1664	register int i;			
1665	for(i = 0; i < pam_no; ++i)			
1666	first_pattern += pac[lane_no].pam_size[i];			
1667	return(first_pattern);			
1668	}			
1669	/*			
1670	int pac_replay()			
1671	/*			
1672	INPUT: none			
1673	OUTPUT: return code = SUCCESS or FAILURE			
1674	DESCRIPTION: Performs a pattern play from pattern			
1675	number entered by user. Assumes play has just been			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pac\_util.c

DATE 5/23/89  
TIME 4:41:22 pm

PAGE #  
15/88

LINE #	SOURCE TEXT
1681	/* performed to get default pattern number.
1682	*/
1683	int
1684	pac_replay()
1685	{
1686	int first_patterns;
1687	int last_block;
1688	int input_good;
1689	
1690	/* Find out where last pattern play left off */
1691	last_block = pacptr(current_lane) -> branch_address;
1692	/* Look up first block number in link table and compute pattern */
1693	first_pattern = (Read_long(pac[current_lane].lane_offset + LINK_OFFSET +
1694	(last_block << 2) + 0x7fff) - BLOCK_SIZE;
1695	do
1696	{
1697	input_good = TRUE;
1698	diag_get_ubox((u_long *) &first_patterns, "first pattern number (hex)", 01,
1699	(u_long)(pac[current_lane].sum_patterns - BLOCK_SIZE));
1700	if((first_patterns & BLOCK_SIZE) != 0)
1701	{
1702	(void)la_message("First patterns must be multiple of 256 (100 hex).\n");
1703	input_good = FALSE;
1704	}
1705	} while(input_good != TRUE);
1706	
1707	pac_set_first_block(Lane_code(current_lane), first_patterns / BLOCK_SIZE);
1708	if(diag_play() != SUCCESS)
1709	{
1710	(void)la_error("Pattern play failed trying to play from pattern %X.\n",
1711	first_patterns);
1712	return(FAILURE);
1713	}
1714	return(SUCCESS);
1715	}

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/parity.c	DATE 5/23/89	PAGE # 1/89
LINE #	SOURCE TEXT			
1	/* SCCS_ID: parity.c rev 3.1, 4/24/89 at 07:49:32 */			
2	/*			
3	** Parity error handler			
4	**			
5	*/			
6	#include "common.h"			
7	#include "cpu.h"			
8	#include "mod_err.h"			
9	#include "svram.h"			
10	#include "lm_rd_wr.h"			
11	extern u_short lm_svram_access();			
12	extern void output_routine(), reset_cpu();			
13	parity_error()			
14	{			
15	register cpu_control_reg_struct *control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
16	register cpu_par_err_reg *parity_reg = (cpu_par_err_reg *) CPU_PAR_ERR_REG, parity;			
17	u_long error;			
18	char buffer[ 100 ];			
19	parity = *parity_reg;			
20	if (control_reg->not_parity_intr == 0)			
21	{			
22	sprintf(buffer, "Parity error addr = %08x\n",			
23	parity_reg->error_addr + 4);			
24	output_routine(buffer);			
25	sprintf(buffer, "%020 id vme id LANCE id\n",			
26	parity_reg->not_68010_master,			
27	parity_reg->not_vme_master,			
28	parity_reg->not_LANCE_master);			
29	output_routine(buffer);			
30	sprintf(buffer, "hi id um id lm id lo id\n",			
31	parity_reg->not_error_hi,			
32	parity_reg->not_error_um,			
33	parity_reg->not_error_lm,			
34	parity_reg->not_error_lo);			
35	output_routine(buffer);			
36	(void)lm_svram_access((char *) parity_reg, PARITY_ERR, sizeof(PARITY_ERR), MEMORY_WRITE, &error);			
37	control_reg->parity_force= 0;			
38	reset_cpu(PARITY_ERROR);			
39	}			
40	}			
41	u_long network_timeout;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pat\_bus.c

DATE 5/23/89  
TIME 4:41:23 pm

PAGE #  
1/90

```

1  // SCCS ID: pat_bus.c rev 3.1.1, 4/24/89 at 07:49:36
2  //
3  //
4  //
5  // Patterns bus for
6  // used in PAC/TEL diagnostics
7  //
8  //
9  //
10 #include "common.h"
11 #include "vrtx.h"
12 #include "mod_def.h"
13 #include "modeler_extn.h"
14 #include "ls_diags.h"
15 #include "tmg.h"
16 #include "tmg_extn.h"
17 #include "tmg_def.h"
18 #include "pac.h"
19 #include "pac_def.h"
20 #include "pac_extn.h"
21 #include "magic.h"
22 #include "pel.h"
23
24
25 pel_disable_bp_error(lase, pel_no)
26 int lase;
27 int pel_no;
28 {
29     short dummy;
30     register PEL *pel = (PEL *) (pel_addr(lase, pel_no));
31
32     dummy = pel->magic_chip[0].m.reset; /* Clear MAGIC errors */
33     pel->car.bit.reset = 0;
34 #ifdef lint
35     if (dummy); /* shut lint up */
36 #endif lint
37
38
39 int
40 pel_xor_patterns_bits(patterns_no, bad_bits)
41 int patterns_no;
42 u_short *bad_bits;
43 {
44     register PEL *pel = (PEL *) (pel_addr(current_lase, current_pel));
45     register u_short chip;
46     register u_short chip;
47     PAT_WORD patterns_word, *patptr = &patterns_word;
48     int magic_chip;
49     int returncode = SUCCESS;
50     u_short diffs;
51
52     /* Read the patterns in pattern memory */
53     (void) pac_read_pattern(patterns_no, patptr);
54     /* Compare patterns with the outputs of the MAGIC chips one at a time */
55     for (magic_chip = 0; magic_chip < 5; ++magic_chip)
56     {
57         memory = patptr->pattern.data[4 - magic_chip];
58         chip = pel->magic_chip[magic_chip].m.data_format;
59         if ((diffs = memory ^ chip) != 0)
60         {
61             (void) lm_error("MAGIC chip %d has bad bits. Expected %04X. Actual %04X.\n",
62                 magic_chip, memory, chip);
63             bad_bits[4 - magic_chip] |= diffs;
64             returncode = FAILURE;
65         }
66     }
67     return(returncode);
68 }
69
70 #define PLAY_ZEROS "x3x5x5x5x5x5x5x5x5x5x7"
71 #define PLAY_ZEROS_SIZE (sizeof(PLAY_ZEROS) / 2)
72 int
73 patterns_bits_test(patterns_no)
74 register int patterns_no;
75 {
76     register int first_pattern = pac_get_first_pattern_no(current_lase, patterns_no);
77     register int end_pattern;
78     u_short bad_bits[5];
79     register int timeout;
80     register int magic_chip;
81     register int pattern_bit;
82     register int returncode = SUCCESS;
83
84     /* Initialize the current lase for patterns play */
85     if (pel_lase_init() != SUCCESS)
86     {
87         (void) lm_error("Unable to initialize lase for patterns bits test.\n");
88         return(FAILURE);
89     }
90
91     /* Build a pattern string which plays a zero pattern to the PEL */
92     if (build_pattern_data(PLAY_ZEROS, first_pattern) != PLAY_ZEROS_SIZE)
93     {
94         (void) lm_error("Unable to build pattern data for patterns bits test.\n");
95         return(FAILURE);
96     }
97     if (build_patterns_control(first_pattern, PLAY_ZEROS_SIZE, STOP_MODE)
98         != SUCCESS)
99     {
100         (void) lm_error("Unable to build pattern control in patterns bits test.\n");
101         return(FAILURE);
102     }
103     if ((timeout = pac_pre_play()) == 0)
104     {
105         (void) lm_error("Unable to prepare for pattern bit test play.\n");
106         return(FAILURE);
107     }
108     /* Clear error bits */
109     for (magic_chip = 0; magic_chip < 5; ++magic_chip)
110         bad_bits[magic_chip] = 0;
111
112     end_pattern = first_pattern + PLAY_ZEROS_SIZE - 1;
113     for (pattern_bit = 0; pattern_bit < 80; ++pattern_bit)
114     {
115         set_pattern_bit(end_pattern, pattern_bit, 1);
116         /* Clear any MAGIC errors and make sure the PEL can't assert a PLAY error */
117         pel_disable_bp_error(current_lase, current_pel);
118         if (pac_play(timeout) != SUCCESS)
119         {
120             returncode = FAILURE;
121         }
122     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pat_bus.c	DATE 5/23/89 TIME 4:41:23 pm	PAGE # 2/91
LINE #	SOURCE TEXT			
121	if(!m_error("Play failed during pattern bit test.\n") != SUCCESS)			
122	return(FAILURE);			
123	{			
124	if(pel_xor_pattern_bits(end_pattern, bad_bits)			
125	!= SUCCESS)			
126	{			
127	returncode = FAILURE;			
128	if(!m_error("Pattern bits did not match.\n") != SUCCESS)			
129	return(FAILURE);			
130	}			
131	set_pattern_bit(end_pattern, pattern_bit, 0);			
132	}			
133	if(returncode == FAILURE)			
134	{			
135	/* Print out all bad bits */			
136	(void)m_message("Bad pattern bits: ");			
137	for(magic_chip = 0; magic_chip < 5; ++magic_chip)			
138	(void)m_message("%04X", bad_bits[magic_chip]);			
139	(void)m_message("\n");			
140	}			
141	return(returncode);			
142	}			
143	}			
144	}			
145	int			
146	pel_check_for_parity_errors()			
147	{			
148	register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));			
149	register int chip;			
150	register int returncode = SUCCESS;			
151	{			
152	for(chip = 0; chip < 5; ++chip)			
153	{			
154	if(pel->magic_chip[chip].m.parity_out != 0)			
155	{			
156	returncode = FAILURE;			
157	if(!m_error("Magic chip %d detected a parity error.\n", chip) != SUCCESS)			
158	return(FAILURE);			
159	}			
160	}			
161	return(returncode);			
162	}			
163	}			
164	}			
165	#define PEL_0_CTL_0 0x7f1			
166	int			
167	pattern_bus_test(first_pattern, num_patterns)			
168	int first_pattern,			
169	int num_patterns,			
170	{			
171	register long *memptr;			
172	register int load_patterns = num_patterns - 1;			
173	register int pel_no = current_pel;			
174	int returncode = SUCCESS;			
175	{			
176	/* Initialize the current lane for pattern play */			
177	if(pel_lane_init() != SUCCESS)			
178	{			
179	(void)m_error("Unable to initialize lane for pattern bus test.\n");			
180	return(FAILURE);			
181	}			
182	/* Clear any MAGIC errors and make sure the PEL can't assert a PLAY error */			
183	/* on the backplane */			
184	pel_disable_bp_error(current_lane, current_pel);			
185	/* Fill pat_ctrl memory with random data */			
186	if(pec_fill_random(first_pattern, num_patterns, PEL_0_CTL_0, SEED) != SUCCESS)			
187	{			
188	(void)m_error("Unable to fill memory with random data in pattern bus test.\n");			
189	return(FAILURE);			
190	}			
191	/* Now OR in the current PEL and the PEL control bits 010 in the first */			
192	/* location and 100 in the rest */			
193	memptr = (long *) pattern_ctrl_addr(current_lane, 2, first_pattern);			
194	*memptr++  = pel_no   (0x2 << 4);			
195	do			
196	{			
197	*memptr++  = pel_no   (0x4 << 4);			
198	}			
199	while(--load_patterns);			
200	if(build_patterns_control(first_pattern, num_patterns, STOP_MODE) != SUCCESS)			
201	{			
202	(void)m_error("Unable to build pattern control in pattern bus test.\n");			
203	return(FAILURE);			
204	}			
205	if(diag_play() != SUCCESS)			
206	{			
207	(void)m_error("Pattern play failed during pattern bus test.\n");			
208	return(FAILURE);			
209	}			
210	/* Make sure there weren't any lane errors */			
211	if(pec_check_errors(current_lane, m_error) != SUCCESS)			
212	{			
213	Clear_pec_errors(current_lane);			
214	(void)m_error("Pattern play caused a Pattern Controller error.\n");			
215	returncode = FAILURE;			
216	}			
217	if(pel_check_for_parity_errors() != SUCCESS)			
218	{			
219	pel_disable_bp_error(current_lane, current_pel);			
220	(void)m_error("Pattern play caused a Pin Electronics error.\n");			
221	returncode = FAILURE;			
222	}			
223	return(returncode);			
224	}			
225	}			
226	}			
227	}			
228	int			
229	pel_patterns_bus_test()			
230	{			
231	if(diag_clear_errors() != SUCCESS)			
232	return(FAILURE);			
233	}			
234	return(pattern_bus_test(0, PATTERNS_IN_128K));			
235	}			
236	}			
237	}			
238	int			
239	pel_pattern_bits_test()			
240	{			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pat\_bus.c

DATE 5/23/89

PAGE #

TIME 4:41:23 pm

3/92

```

LINE # SOURCE TEXT
241 if(diag_clear_errors() != SUCCESS)
242     return(FAILURE);
243
244     return(pattern_bits_test(0));
245 }
246
247
248 int
249 pel_check_parity(pattern, error_chip)
250 int pattern;
251 int error_chip;
252 {
253     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
254     PAT_WORD pattern_word, *patptr = &pattern_word;
255     long diff;
256     long memory;
257     long sum_bits;
258     long parity;
259     int chip;
260     int i;
261     int returncode = SUCCESS;
262
263     /* Read the pattern with the parity error */
264     (void)pac_read_pattern(pattern, patptr);
265
266     /* Verify that only the proper magic chip detected a parity error */
267     /* and that the data compares in each chip */
268     for(chip = 0; chip < 5; ++chip)
269     {
270         if(chip == error_chip) /* Should have detected error */
271         {
272             if(pel->magic_chip[chip].m.parity_out != 1)
273             {
274                 returncode = FAILURE;
275                 if(!error("Magic chip %d did not detect error.\n", chip) != SUCCESS)
276                     return(FAILURE);
277             }
278         }
279         else /* Should not have detected error */
280         {
281             if(pel->magic_chip[chip].m.parity_out != 0)
282             {
283                 returncode = FAILURE;
284                 if(!error("Magic chip %d detected error.\n", chip) != SUCCESS)
285                     return(FAILURE);
286             }
287         }
288     }
289     /* Read the word stored in memory and compute odd parity */
290     memory = patptr->pattern_data[4 - chip];
291     for(sum_bits = 0, i = 0; i < 16; ++i)
292         sum_bits += (memory >> i) & 1;
293     parity = sum_bits & 1;
294     if(pel->magic_chip[chip].m.parity_in != ((chip == error_chip) ?
295        ('Parity') & 1 : parity))
296     {
297         returncode = FAILURE;
298         if(!error("Magic chip %d latched incorrect parity sense.\n", chip)
299            != SUCCESS)
300             return(FAILURE);
301     }
302     if((diff = pel->magic_chip[chip].m.error_data - memory) != 0)
303     {
304         returncode = FAILURE;
305         if(!error("Magic chip %d latched the following bad bits: %04X.\n",
306            chip, diff) != SUCCESS)
307             return(FAILURE);
308     }
309 }
310
311     return(returncode);
312 }
313
314 #define PEL_ERROR_DELAY 6
315
316 /*
317  * int pel_parity_error_test()
318  *
319  * INPUT: none
320  * OUTPUT: returncode = SUCCESS or FAILURE
321  * DESCRIPTION:
322  */
323 int
324 pel_parity_error_test()
325 {
326     register int pel_no = current_pel;
327     register long *memptr;
328     int pattern;
329     int timeout;
330     int branch_address[3];
331     int block_offset[3];
332     int actual_branch;
333     int actual_block;
334     int magic_chip;
335     int i;
336     int returncode = SUCCESS;
337
338     if(diag_clear_errors() != SUCCESS)
339         return(FAILURE);
340
341     /* Initialize the current lane for pattern play */
342     if(pel_lane_init() != SUCCESS)
343     {
344         (void)la_error("Unable to initialize lane for pattern bits test.\n");
345         return(FAILURE);
346     }
347
348     if(dab_type == NO_DAB) /* Make sure we don't get a play error */
349         pel_disable_bp_error(current_lane, current_pel);
350     else
351         pel_dab_init(PUBLIC_DAB);
352
353     /* Fill a block of pattern memory with random data, starting with block 1 */
354     if(pac_fill_random(BLOCK_SIZE, BLOCK_SIZE, PEL_0_CTL_0, SEED) != SUCCESS)
355     {
356         (void)la_error("Could not fill block number one with random data.\n");
357         return(FAILURE);
358     }
359
360     /* Now OR in the current PEL and the PEL control bits 010 in the first */
361     /* location and 100 in the rest */
362     memptr = (long *)Pattern_to_address(current_lane, 2, BLOCK_SIZE);

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pat\_bus.c

DATE 5/23/89 PAGE #  
TIME 4:41:23 pm 4/93

LINE #	SOURCE TEXT
361	/*memptr++  = pel_no   (dab << 4);
362	pattern = BLOCK_SIZE - 1;
363	do
364	{
365	/*memptr++  = pel_no   (dab << 4);
366	}
367	while(--pattern);
368	if(build_pattern_control(BLOCK_SIZE, BLOCK_SIZE, STOP_MODE) != SUCCESS)
369	{
370	(void)lm_error("Could not build pattern control in block number one.\n");
371	return(FAILURE);
372	}
373	/* Prepare for pattern play and compute a conservative timeout */
374	if((timeout = pec_pre_play()) == 0)
375	{
376	(void)lm_error("Pattern play preparation failed.\n");
377	return(FAILURE);
378	}
379	for(pattern = BLOCK_SIZE - 1; pattern < ((2 * BLOCK_SIZE) - 1); ++pattern)
380	{
381	if(dab_type != NO_DAB)
382	{
383	for(i = 0; i < 3; ++i)
384	{
385	if(pec_predict_offset(pattern, PEL_ERROR_DELAY + 1, &branch_address[i],
386	&block_offset[i]) != SUCCESS)
387	{
388	(void)lm_error("Unable to predict parity error offsets for pattern \
389	number %d, error delay = %d.\n", pattern, PEL_ERROR_DELAY + 1);
390	return(FAILURE);
391	}
392	}
393	}
394	for(magic_chip = 0; magic_chip < 5; ++magic_chip)
395	{
396	if(pec_insert_parity_error(pattern, magic_chip + 1) != SUCCESS)
397	{
398	(void)lm_error("Could not insert parity error for magic chip %d.\n",
399	magic_chip);
400	return(FAILURE);
401	}
402	pecptr[current_lane]-->branch_address = 1;
403	if(pec_play(timeout) != SUCCESS)
404	{
405	(void)lm_error("Pattern play failed.\n");
406	return(FAILURE);
407	}
408	/* If there's a DAB, see if pattern play generated a backplane error */
409	if((dab_type != NO_DAB) && ((Get_bp_error() & lane_code(current_lane))
410	== 0))
411	{
412	(void)lm_error("Pattern play did not generate error (should have).\n");
413	return(FAILURE);
414	}
415	if(pec_check_parity(pattern, magic_chip) != SUCCESS)
416	{
417	returncode = FAILURE;
418	if(lm_error("Magic chip parity error verification failed.\n")
419	!= SUCCESS)
420	return(FAILURE);
421	}
422	/* Check PAC offsets if there is a DAB */
423	if(dab_type != NO_DAB)
424	{
425	actual_branch = pecptr[current_lane]-->branch_address;
426	if((actual_branch != branch_addresses[0]) &&
427	(actual_branch != branch_addresses[1]) &&
428	(actual_branch != branch_addresses[2]))
429	{
430	returncode = FAILURE;
431	if(lm_error("Branch address for pattern %d is incorrect.\n\
432	tExpected %d, %d or %d. Actual %d.\n", pattern, branch_addresses[0],
433	branch_addresses[1], branch_addresses[2], actual_branch) != SUCCESS)
434	return(FAILURE);
435	}
436	actual_block = pecptr[current_lane]-->block_offset;
437	if((actual_block != block_offset[0]) &&
438	(actual_block != block_offset[1]) &&
439	(actual_block != block_offset[2]))
440	{
441	returncode = FAILURE;
442	if(lm_error("Block offset for pattern %d is incorrect.\n\
443	tExpected %d, %d or %d. Actual %d.\n", pattern, block_offset[0],
444	block_offset[1], block_offset[2], actual_block) != SUCCESS)
445	return(FAILURE);
446	}
447	}
448	if(pec_remove_parity_error(pattern, magic_chip + 1) != SUCCESS)
449	{
450	(void)lm_error("Could not remove parity error for magic chip %d.\n",
451	magic_chip);
452	return(FAILURE);
453	}
454	/* Get rid of any PEL errors */
455	if(dab_type == NO_DAB)
456	pel_disable_bp_error(current_lane, current_pel);
457	else
458	pel_clear_pel_errors();
459	}
460	}
461	return(returncode);
462	}



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel.c

DATE 5/23/89  
TIME 4:41:24 pm

PAGE #  
1/94

```

1  /* SCCS ID: pel.c rev 3.1, 4/24/89 at 07:49:40 */
2  /*****
3   * pel_menu.c
4   *
5   * Main Menu
6   * used in PEL diagnostics
7   *
8   *****/
9  /*****
10   * include <math.h>
11   * include "common.h"
12   * include "lm_diags.h"
13   * include "vrtx.h"
14   * include "zmq.h"
15   * include "mod_def.h"
16   * include "modeler_extn.h"
17   * include "magic.h"
18   * include "pel.h"
19   * include "id.h"
20   * include "eeprom.h"
21   *****/
22  int current_pel = 0;
23  int dab_type = NO_DAB;
24
25  #define PEL_TEST_INDEX 0
26  #define PEL_DIAG_DAB_TEST_INDEX 1
27
28  pel_menu(parent_menu, info)
29  LM_DIAG_MENU *parent_menu;
30  char *info;
31  {
32      char buffer[132];
33      static char test_buffer[132];
34      static char diag_dab_test_buffer[132];
35      register char *buf;
36
37      int pel_test_menu();
38      int pel_diag_dab_test_menu();
39      int pel_test_utilities();
40      int pel_play_utilities();
41      int pel_patterns_utilities();
42      extern int lm_acceptance;
43
44      static LM_DIAG_MENU_ITEM menu_list[] =
45      {
46          {
47              "1",
48              test_buffer,
49              pel_test_menu,
50              LM_DIAG_another_menu,
51              LM_DIAG_null,
52              0
53          },
54          {
55              "2",
56              diag_dab_test_buffer,
57              pel_diag_dab_test_menu,
58              0,
59              LM_DIAG_null,
60              0
61          },
62          {
63              "3",
64              "PEL Test Utilities",
65              pel_test_utilities,
66              LM_DIAG_utility_menu,
67              LM_DIAG_null,
68              0
69          },
70          {
71              "4",
72              "PEL Pattern Utilities",
73              pel_patterns_utilities,
74              LM_DIAG_utility_menu,
75              LM_DIAG_null,
76              0
77          },
78          {
79              "5",
80              "PEL Play Utilities",
81              pel_play_utilities,
82              LM_DIAG_utility_menu,
83              LM_DIAG_null,
84              0
85          },
86      };
87
88      static LM_DIAG_MENU menu =
89      {
90          0,
91          sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
92          0,
93          menu_list
94      };
95
96      /*****
97       *
98       * current_pel = (long)(info - '0');
99       * if (Host) {
100       *     dab_type = DIAG_DAB;
101       * } else {
102       *     dab_type = what_dab(current_lane, current_pel);
103       * }
104       *
105       * if (lm_acceptance && (dab_type != DIAG_DAB))
106       *     return SUCCESS;
107       *
108       * menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes
109       *     = LM_DIAG_another_menu | LM_DIAG_disable;
110       * switch (dab_type) {
111       *     case NO_DAB:
112       *         buf = "NO";
113       *         break;
114       *     case DIAG_DAB:
115       *         buf = "DIAGNOSTIC";
116       *         menu_list[PEL_TEST_INDEX].attributes
117       *             |= LM_DIAG_acceptance;
118       *         menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes
119       *             |= LM_DIAG_acceptance;
120       *         /* turn off the disable bit only if there is a PAC

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel.c

DATE 5/23/89  
TIME 4:41:24 pm

PAGE #  
2/95

LINE #	SOURCE TEXT
121	in the lane */
122	if(probe_poc(current_lane) == SUCCESS)
123	{
124	menu_list[PEL_DIAG_DAB_TEST_INDEX].attributes
125	= LM_DIAG_disable;
126	}
127	break;
128	case UNKNOWN_DAB:
129	default:
130	buf = "NON-DIAGNOSTIC";
131	break;
132	}
133	
134	sprintf(buf, "%s (%s DAB)", parent_menu->
135	menu_items[parent_menu->current_selection].menu_text, buf);
136	menu.title = buf;
137	
138	sprintf(test_buffer, "Lane to PEL to Test Menu (%s DAB)",
139	current_lane + 'A', current_pel, buf);
140	
141	sprintf(diag_dab_test_buffer,
142	"Lane to PEL to Diagnostic Adapter Test Menu (%s DAB)",
143	current_lane + 'A', current_pel, buf);
144	
145	return lm_display_menus(&menu);
146	
147	
148	pel_test_menu(parent_menu)
149	LM_DIAG_MENU *parent_menu;
150	{
151	register long i;
152	register int pec_present;
153	register LM_DIAG_MENU_ITEM *m;
154	
155	int pel_idprom_checksum();
156	int pel_wr_car();
157	int pel_wr_magic_ctl();
158	int pel_dac_adc();
159	int pel_patterns_control();
160	int pel_patterns_bits_test();
161	int pel_patterns_bus_test();
162	int pel_parity_error_test();
163	int pel_edge();
164	
165	static LM_DIAG_MENU_ITEM menu_list[] =
166	{
167	{
168	"1",
169	"Pis Electronics ID Prom Test",
170	pel_idprom_checksum,
171	LM_DIAG_diag_routine LM_DIAG_acceptance,
172	LM_DIAG_wall,
173	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB)
174	},
175	{
176	"2",
177	"Control/Status Register Test",
178	pel_wr_car,
179	LM_DIAG_diag_routine LM_DIAG_acceptance,
180	LM_DIAG_wall,
181	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB)
182	},
183	{
184	"3",
185	"DACs and ADCs Test",
186	pel_dac_adc,
187	LM_DIAG_diag_routine LM_DIAG_acceptance,
188	LM_DIAG_wall,
189	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB)
190	},
191	{
192	"4",
193	"Magic Chip Control Register Test",
194	pel_wr_magic_ctl,
195	LM_DIAG_diag_routine LM_DIAG_acceptance,
196	LM_DIAG_wall,
197	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
198	},
199	{
200	"5",
201	"Patterns Control Test",
202	pel_patterns_control,
203	LM_DIAG_diag_routine LM_DIAG_acceptance,
204	LM_DIAG_wall,
205	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
206	},
207	{
208	"6",
209	"Patterns Bit Test",
210	pel_patterns_bits_test,
211	LM_DIAG_diag_routine LM_DIAG_acceptance,
212	LM_DIAG_wall,
213	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
214	},
215	{
216	"7",
217	"Patterns Bus Test",
218	pel_patterns_bus_test,
219	LM_DIAG_diag_routine LM_DIAG_acceptance,
220	LM_DIAG_wall,
221	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
222	},
223	{
224	"8",
225	"Parity Error Detection Test",
226	pel_parity_error_test,
227	LM_DIAG_diag_routine LM_DIAG_acceptance,
228	LM_DIAG_wall,
229	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
230	},
231	{
232	"9",
233	"Timing Edge Test",
234	pel_edge,
235	LM_DIAG_diag_routine LM_DIAG_acceptance,
236	LM_DIAG_wall,
237	(char *) (DIAG_DAB NO_DAB UNKNOWN_DAB NEED_PAC)
238	},
239	};
240	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel.c

DATE 5/23/89

PAGE #

TIME 4:41:24 pm

3/96

```

LINE # SOURCE TEXT
241 static LM_DIAG_MENU menu =
242 {
243     0,
244     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
245     0,
246     menu_list
247 },
248
249 /*=====*/
250
251 if (Host) {
252     dab_type = DIAG_DAB,
253 } else {
254     dab_type = what_dab(current_lane, current_pel);
255     pac_present = probe_pac(current_lane);
256     for (m = menu_list, i = 0; i < menu.number_of_items; ++i, ++m) {
257         if (((long)(m->user_data) & dab_type) {
258             m->attributes |= LM_DIAG_disable;
259         } else {
260             m->attributes |= LM_DIAG_disable;
261             if (((long)(m->user_data) & NEED_PAC) != 0) &&
262                 (pac_present != SUCCESS)
263                 m->attributes |= LM_DIAG_disable;
264         }
265     }
266
267     menu.title = parent_menu->
268     menu_items[parent_menu->current_selection].menu_text,
269     return lm_display_menu(menu);
270 }
271
272 pel_diag_dab_test_menu(parent_menu)
273 LM_DIAG_MENU *parent_menu;
274
275 register long i;
276 register LM_DIAG_MENU_ITEM *m;
277
278 int pel_opens_and_shorts();
279 int pel_crc();
280 int pel_feedback_test();
281 int pel_short_sensor_test();
282 int pel_error_test();
283 int pel_keeplive_test();
284 int pel_driver_test();
285 int pel_comparator_test();
286
287 static LM_DIAG_MENU_ITEM menu_list[] =
288 {
289     {
290         "1",
291         "Diagnostic Adapter Opens/Shorts Test",
292         pel_opens_and_shorts,
293         LM_DIAG_diag_routine|LM_DIAG_acceptance,
294         LM_DIAG_null,
295         (char *) (DIAG_DAB)
296     },
297     {
298         "2",
299         "Diagnostic Adapter CRC Test",
300         pel_crc,
301         LM_DIAG_diag_routine|LM_DIAG_acceptance,
302         LM_DIAG_null,
303         (char *) (DIAG_DAB)
304     },
305     {
306         "3",
307         "Diagnostic Adapter Feedback Test",
308         pel_feedback_test,
309         LM_DIAG_diag_routine|LM_DIAG_acceptance,
310         LM_DIAG_null,
311         (char *) (DIAG_DAB)
312     },
313     {
314         "4",
315         "Diagnostic Adapter Short Sensor Test",
316         pel_short_sensor_test,
317         LM_DIAG_diag_routine|LM_DIAG_acceptance,
318         LM_DIAG_null,
319         (char *) (DIAG_DAB)
320     },
321     {
322         "5",
323         "Diagnostic Adapter Error Handling Test",
324         pel_error_test,
325         LM_DIAG_diag_routine|LM_DIAG_acceptance,
326         LM_DIAG_null,
327         (char *) (DIAG_DAB)
328     },
329     {
330         "6",
331         "Diagnostic Adapter Keelalive/Trigger Bit Test",
332         pel_keeplive_test,
333         LM_DIAG_diag_routine|LM_DIAG_acceptance,
334         LM_DIAG_null,
335         (char *) (DIAG_DAB)
336     },
337     {
338         "7",
339         "Diagnostic Adapter LM-1000 Driver Test",
340         pel_driver_test,
341         LM_DIAG_diag_routine|LM_DIAG_acceptance,
342         LM_DIAG_null,
343         (char *) (DIAG_DAB)
344     },
345     {
346         "8",
347         "Diagnostic Adapter LM-1000 Receiver Test",
348         pel_comparator_test,
349         LM_DIAG_diag_routine|LM_DIAG_acceptance,
350         LM_DIAG_null,
351         (char *) (DIAG_DAB)
352     },
353     {
354         "9",
355         "ADD THESE TESTS",
356         0,
357         0,
358         0,
359         0
360     }
361 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel.c

DATE 5/23/89  
TIME 4:41:24 pm

PAGE #  
4/97

```

LINE # SOURCE TEXT
361 //
362
363
364 static LM_DIAG_MENU menu =
365 {
366     0,
367     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
368     0,
369     menu_list
370 },
371
372 /*****
373
374 if (Host) {
375     dab_type = DIAG_DAB;
376 } else {
377     dab_type = what_dab(current_lane, current_pel);
378     for (m = menu_list, i = 0; i < menu.number_of_items; ++i, ++m) {
379         if ((!loop) & m->user_data) & dab_type) {
380             m->attributes |= LM_DIAG_disable;
381         } else {
382             m->attributes |= LM_DIAG_disable;
383         }
384     }
385 }
386
387 menu.title = parent_menu->
388 menu.items[parent_menu->current_selection].menu_text;
389 return lm_display_menu(menu);
390 }
391
392 pel_test_utilities(parent_menu)
393 LM_DIAG_MENU *parent_menu;
394
395 int pel_type_patterns_string();
396 int display_magic_control();
397 int pel_set_pel_ctrl();
398 show_all_adc();
399 int loop_adc_read();
400 int loop_dac_toggle();
401 int display_patterns_memory();
402 int display_patterns_preamble();
403 int display_patterns_preamble_verbose();
404 int print_magic_status();
405 int display_pel_errors();
406
407 static LM_DIAG_MENU_ITEM menu_list[] =
408 {
409     {
410         "1",
411         "Display and set pel control register",
412         pel_set_pel_ctrl,
413         LM_DIAG_utility,
414         LM_DIAG_null,
415         0
416     },
417     {
418         "2",
419         "Display MAGIC control register",
420         display_magic_control,
421         LM_DIAG_utility,
422         LM_DIAG_null,
423         0
424     },
425     {
426         "3",
427         "Display MAGIC status register",
428         print_magic_status,
429         LM_DIAG_utility,
430         LM_DIAG_null,
431         0
432     },
433     {
434         "4",
435         "Display PEL errors",
436         display_pel_errors,
437         LM_DIAG_utility,
438         LM_DIAG_null,
439         0
440     },
441     {
442         "5",
443         "Show all ADC values",
444         show_all_adc,
445         LM_DIAG_utility,
446         LM_DIAG_null,
447         0
448     },
449     {
450         "6",
451         "Read ADC indefinitely",
452         loop_adc_read,
453         LM_DIAG_utility,
454         LM_DIAG_null,
455         0
456     },
457     {
458         "7",
459         "Toggle DAC outputs indefinitely",
460         loop_dac_toggle,
461         LM_DIAG_utility,
462         LM_DIAG_null,
463         0
464     },
465 },
466
467 static LM_DIAG_MENU menu =
468 {
469     "PEL UTILITIES",
470     sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
471     0,
472     menu_list
473 },
474
475 menu.title = parent_menu->
476 menu.items[parent_menu->current_selection].menu_text;
477 return lm_display_menu(menu);
478
479 pel_check_csr(csr, value)
480 register unsigned short *csr;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel.c

DATE 5/23/89

PAGE #

TIME 4:41:24 pm

5/98

```

LINE # SOURCE TEXT
481 u_short value;
482 {
483     *car = value;
484     if ((*car & 0xff) != value) {
485         lm_message("Host 402X, Read 402X\n", value, (*car & 0xff));
486         return (-1);
487     } else {
488         return (0);
489     }
490 }
491
492 pel_set_pel_car()
493 {
494     long answer;
495     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
496
497     lm_message("Pel car = 404X\n", pel->car.reg);
498     lm_message("magic_errorL (td)\n", pel->car.bit.magic_errorL);
499     lm_message("play_errorL (td)\n", pel->car.bit.play_errorL);
500     lm_message("errorL (td)\n", pel->car.bit.errorL);
501     lm_message("present (td)\n", pel->car.bit.present);
502     lm_message("active (td)\n", pel->car.bit.active);
503     lm_message("eeprom_out (td)\n", pel->car.bit.eeprom_out);
504
505     answer = pel->car.bit.eeprom_sel;
506     diag_get_long(answer, "eeprom_sel", 01, 11);
507     pel->car.bit.eeprom_sel = answer;
508
509     answer = pel->car.bit.eeprom_clk;
510     diag_get_long(answer, "eeprom_clk", 01, 11);
511     pel->car.bit.eeprom_clk = answer;
512
513     answer = pel->car.bit.eeprom_in;
514     diag_get_long(answer, "eeprom_in", 01, 11);
515     pel->car.bit.eeprom_in = answer;
516
517     answer = pel->car.bit.initialize;
518     diag_get_long(answer, "initialize", 01, 11);
519     pel->car.bit.initialize = answer;
520
521     answer = pel->car.bit.magic_error_enableL;
522     diag_get_long(answer, "magic_error_enableL", 01, 11);
523     pel->car.bit.magic_error_enableL = answer;
524
525     answer = pel->car.bit.private;
526     diag_get_long(answer, "private", 01, 11);
527     pel->car.bit.private = answer;
528
529     answer = pel->car.bit.resetL;
530     diag_get_long(answer, "resetL", 01, 11);
531     pel->car.bit.resetL = answer;
532
533     answer = pel->car.bit.in_use_led;
534     diag_get_long(answer, "in_use_led", 01, 11);
535     pel->car.bit.in_use_led = answer;
536 }
537
538 pel_wr_car()
539 {
540     register PEL *pel
541         = (PEL *) (pel_addr(current_lane, current_pel));
542     register unsigned short *car = &(pel->car.reg);
543
544     register long errors = 0;
545
546     if (!Host) {
547         errors += pel_check_car(car, 0x00);
548         errors += pel_check_car(car, 0x01);
549         errors += pel_check_car(car, 0x02);
550         errors += pel_check_car(car, 0x04);
551         errors += pel_check_car(car, 0x08);
552         errors += pel_check_car(car, 0x10);
553         errors += pel_check_car(car, 0x20);
554         errors += pel_check_car(car, 0x40);
555         errors += pel_check_car(car, 0x80);
556         errors += pel_check_car(car, 0x00);
557     } else {
558         lm_message("Can't run this test on the host\n");
559     }
560
561     lm_message("lane tc / pel id / address tx / ", current_lane + 'A', current_pel, car);
562     lm_message("PEL size tx\n", sizeof(PEL));
563     pel_display_car(pel);
564
565     if (errors) {
566         return(FAILURE);
567     } else {
568         return(SUCCESS);
569     }
570 }
571
572 pel_display_car(pel)
573 {
574     register PEL *pel;
575
576     lm_message("PEL CSR(404X):\n", pel->car.reg);
577     lm_message("MAGICERROR = td\n", pel->car.bit.magic_errorL);
578     lm_message("PLAYERROR = td\n", pel->car.bit.play_errorL);
579     lm_message("ERROR = td\n", pel->car.bit.errorL);
580     lm_message("PRESENT = td\n", pel->car.bit.present);
581     lm_message("ACTIVE = td\n", pel->car.bit.active);
582     lm_message("EEDOUT = td\n", pel->car.bit.eeprom_out);
583     lm_message("EESSEL = td\n", pel->car.bit.eeprom_sel);
584     lm_message("ECLK = td\n", pel->car.bit.eeprom_clk);
585     lm_message("EIN = td\n", pel->car.bit.eeprom_in);
586     lm_message("INITIALIZE = td\n", pel->car.bit.initialize);
587     lm_message("MAGIC ERROR = td\n", pel->car.bit.magic_error_enableL);
588     lm_message("PRIVATE = td\n", pel->car.bit.private);
589     lm_message("RESET = td\n", pel->car.bit.resetL);
590     lm_message("INUSE LED = td\n", pel->car.bit.in_use_led);
591 }
592
593 pel_idprom_checksum()
594 {
595     register PEL *pel
596         = (PEL *) (pel_addr(current_lane, current_pel));
597     ID_PROM PEL id;
598
599     if (!Host) {
600         lm_message("Can't read pel id prom on host\n");
        return FAILURE;
    }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel.c	DATE 5/23/89	PAGE # 6/99
LINE #	SOURCE TEXT			
601	}			
602	}			
603	}			
604	pel_id_load(pel, tid);			
605	lm_message("\aPEL base address = %08x\n", pel);			
606	lm_message("Board type = %04d\n", id.generic.board_type);			
607	lm_message("Revision = %c\n", id.generic.revision);			
608	lm_message("ECO level = %d\n", id.generic.eco_level);			
609	lm_message("Speed = %04x\n", id.speed);			
610	lm_message("Pattern width = %d\n", id.width * ID_GEN_WIDTH_K);			
611	lm_message("Pattern memory = %dK\n", id.pattmem);			
612	lm_message("MAGIC fab date = %02d%02d\n", id.magic_build.year, id.magic_build.week);			
613	lm_message("Checksum = %02x\n", id.checksum);			
614	}			
615	if (id.generic.board_type != ID_BT_TILPEL) {			
616	(void)pel_error("ID From is not TILPEL\n");			
617	return(FAILURE);			
618	}			
619	}			
620	if (!pel_id_check(pel)) {			
621	return SUCCESS;			
622	} else {			
623	return FAILURE;			
624	}			
625	}			
626	}			
627	what_dab(lane, pel_no)			
628	int lane;			
629	int pel_no;			
630	{			
631	register PEL *pel = (PEL *) (pel_addr(lane, pel_no));			
632	DAB EEPROM dab;			
633	if (Host) {			
634	lm_message("Can't probe on host\n");			
635	return UNKNOWN_DAB;			
636	}			
637	pel->car.reg = 0; /* make sure ERM=0 (for Diagnostic DAB) */			
638	if (!pel->car.bit.present) {			
639	dab_type = NO_DAB;			
640	return(NO_DAB);			
641	} else {			
642	if ((!lm_read_eepram(pel_no + (lane << 3), &dab)			
643	== SUCCESS) && !strcmp(dab.device_name, "DIAGNOSTIC_ADAPTER")) {			
644	dab_type = DIAG_DAB;			
645	return(DIAG_DAB);			
646	} else {			
647	dab_type = UNKNOWN_DAB;			
648	return(UNKNOWN_DAB);			
649	}			
650	}			
651	}			
652	}			
653	}			
654	}			
655	}			
656	}			
657	}			
658	loop_adc_read()			
659	{			
660	register PEL *pel			
661	= (PEL *) (pel_addr(current_lane, current_pel));			
662	int dummy;			
663	flush_key_buf();			
664	lm_message("Hit key to stop...");			
665	while (!lm_check_key()) {			
666	dummy = pel->vwith_adc;			
667	lm_delay(1);			
668	dummy = pel->vwith_adc;			
669	lm_delay(1);			
670	}			
671	return dummy; /* abort list up */			
672	}			
673	}			
674	}			
675	}			
676	}			
677	loop_dec_toggle()			
678	{			
679	register PEL *pel			
680	= (PEL *) (pel_addr(current_lane, current_pel));			
681	flush_key_buf();			
682	lm_message("Hit key to stop...");			
683	while (!lm_check_key()) {			
684	pel->vlogl_dec = 0;			
685	pel->vlogh_dec = 0;			
686	pel->vwith_dec = 0;			
687	pel->val_dec = 0;			
688	pel->vwith_dec = 0;			
689	pel->vwhh_dec = 0;			
690	lm_delay(1);			
691	pel->vlogl_dec = 0x0f;			
692	pel->vlogh_dec = 0x0f;			
693	pel->vwith_dec = 0x0f;			
694	pel->val_dec = 0x0f;			
695	pel->vwith_dec = 0x0f;			
696	pel->vwhh_dec = 0x0f;			
697	lm_delay(1);			
698	}			
699	}			
700	}			
701	}			
702	}			
703	}			
704	}			
705	display_eepram_info()			
706	{			
707	DAB EEPROM dab;			
708	char configbuf[256];			
709	if (!lm_read_eepram(current_pel + (current_lane << 3), &dab) != SUCCESS) {			
710	(void)pel_error("Cannot read DAB EEPROM\n");			
711	return(FAILURE);			
712	}			
713	construct_cfg_str(configbuf, dab.configuration);			
714	lm_message(" Device Adapter in Lane %c Slot %d:\n",			
715	current_lane + 'A', current_pel);			
716	lm_message(" signature: %20.4s  ", dab.signature);			
717	lm_message(" insertion count: %u\n", dab.insertion_count);			
718	}			
719	}			
720	}			

[illegible]

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel.c	DATE 5/23/89 TIME 4:41:24 pm	PAGE # 8/101
LINE #	SOURCE TEXT			
841	int pel_type_pattern_string();			
842	int pel_set_timing();			
843	int pel_set_voltspecs();			
844	int pel_replay();			
845	int pel_loop_play();			
846	int pel_eval_soft_drivers();			
847	int pel_eval_dacs_adcs();			
848	static LM_DIAG_MENU_ITEM menu_list[] =			
849	{			
850	{			
851	{			
852	"1",			
853	"Enter and play patterns",			
854	pel_type_pattern_string,			
855	LM_DIAG_utility,			
856	LM_DIAG_null,			
857	0			
858	},			
859	{			
860	{			
861	"2",			
862	"Set Timing",			
863	pel_set_timing,			
864	LM_DIAG_utility,			
865	LM_DIAG_null,			
866	0			
867	},			
868	{			
869	{			
870	"3",			
871	"Set Voltspecs",			
872	pel_set_voltspecs,			
873	LM_DIAG_utility,			
874	LM_DIAG_null,			
875	0			
876	},			
877	{			
878	{			
879	"4",			
880	"Play",			
881	pel_replay,			
882	LM_DIAG_utility,			
883	LM_DIAG_null,			
884	0			
885	},			
886	{			
887	{			
888	"5",			
889	"Loop play",			
890	pel_loop_play,			
891	LM_DIAG_utility,			
892	LM_DIAG_null,			
893	0			
894	},			
895	{			
896	{			
897	"6",			
898	"Evaluate Soft-Drivers",			
899	pel_eval_soft_drivers,			
900	LM_DIAG_utility,			
901	LM_DIAG_null,			
902	0			
903	},			
904	{			
905	{			
906	"7",			
907	"Evaluate DACs and ADCs",			
908	pel_eval_dacs_adcs,			
909	LM_DIAG_utility,			
910	LM_DIAG_null,			
911	0			
912	},			
913	},			
914	static LM_DIAG_MENU menu =			
915	{			
916	"PEL UTILITIES",			
917	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),			
918	0,			
919	menu_list			
920	};			
921	menu.title = parent_menu->			
922	menu.items[parent_menu->current_selection].menu_text,			
923	return lm_display_menu(menu);			
924	}			
925	display_pel_errors()			
926	{			
927	if (pel_check_errors(current_lase, current_pel, lm_message) == SUCCESS) {			
928	lm_message("No errors on P1a Electronics Module %ld\n", current_pel);			
929	}			
930	return(SUCCESS);			
931	}			
932	pel_check_errors(lase, slot, prtica)			
933	int lase, slot;			
934	int (*prtica)();			
935	{			
936	register PEL *pel = (PEL *) (pel_addr(lase, slot));			
937	int errors = 0;			
938	int i;			
939	if ((pel->car.bit.error) {			
940	if (pel->car.bit.present && (pel->car.bit.active) {			
941	prtica("PEL error: DAB inserted\n");			
942	errors++;			
943	}			
944	if ((pel->car.bit.present && pel->car.bit.initialize) {			
945	prtica("PEL error: DAB removed\n");			
946	errors++;			
947	}			
948	if ((pel->car.bit.play_error) {			
949	prtica("PEL error: Playing to an uninitialized PEL\n");			
950	errors++;			
951	}			
952	if ((pel->car.bit.magic_error) && (pel->car.bit.magic_error_enable) {			
953	for (i = 0; i < 5; i++) {			
954	if (pel->magic_chip[i].m.parity_out) {			
955	prtica("PEL error: MAGIC(%ld) parity error\n", i);			
956	errors++;			
957	}			
958	}			
959	}			
960	}			



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel.c	DATE 5/23/89	PAGE # 9/102
LINE #		SOURCE TEXT		
961		for (i = 0; i < 5; i++) {	/* MAGIC shorts */	
962		if (pel->magic_chip[i].m.short_sample) {		
963		printf("PEL error: MAGIC(%ld) short error = %04x\n",		
964		i, pel->magic_chip[i].m.short_sample);		
965		errors++;		
966		}		
967				
968		if (!errors) {	/* unknown */	
969		printf("PEL error: unknown error\n");		
970		errors++;		
971		}		
972		}		
973				
974		return errors ? FAILURE : SUCCESS;		
975		}		
976				
977				
978				
979		pel_count_dabs()		
980		{		
981		int lane_no, slot_no;		
982		int count = 0;		
983				
984		if (diag_tme_reset(TRUE) != SUCCESS) {		
985		(void)pel_error("Backplane reset failed during lane probe.\n");		
986		return 0;		
987		}		
988		for (lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++) {		
989		if (probe_pac(lane_no) == SUCCESS) {		
990		for (slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++) {		
991		if (probe_pel(lane_no, slot_no) == SUCCESS) {		
992		if (wast_dab(lane_no, slot_no) == DIAG_DAB) {		
993		++count;		
994		}		
995		}		
996		}		
997				
998		return count;		
999		}		
1000				
1001				
1002		pel_error(s)		
1003		char *s;		
1004		{		
1005		(void)ln_error("Lane %d Pel %d: %s",		
1006		current_lane + 'A', current_pel, s);		
1007		}		
1008		}		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_analog.c

DATE	5/23/89	PAGE #
TIME	4:41:25 pm	1/103

```

SOURCE TEXT
LINE #
1  /* SCCS ID: pel_analog.c rev 3.1, 4/24/89 at 07:49:46 */
2
3  .....
4  pel_analog.c
5  .....
6  PEL DAC/ADC tests
7  used in PEL diagnostics
8  .....
9
10 #include <math.h>
11 #include "common.h"
12 #include "lm_diags.h"
13 #include "vtr.h"
14 #include "tmg.h"
15 #include "sod_def.h"
16 #include "modeler_math.h"
17 #include "magic.h"
18 #include "pel.h"
19
20 int set_vlogl_dac();
21 int set_vlogh_dac();
22 int set_vthh_dac();
23 int set_vah_dac();
24 int set_vlth_dac();
25 int set_val_dac();
26
27 #define DUTVCC_min 3.0
28 #define DUTVCC_max 5.25
29 #define VTH_NO_DAB 0.6
30 #define DUT_ANALOG_ERROR 0.2
31 #define ADC_LSB (5.0 / 256.0)
32 #define DUTVCC_ADC_LSB (6.0 / 256.0)
33 #define DAC_ADC_ERROR 0.140
34
35 /* The following formulas convert floating point voltage setting boundaries */
36 /* based on a floating point DUT Vcc value */
37 /* 1 - DUT VCC (float) */
38 #define VLOGL_max(X) (X)
39 #define VLOGH_max(X) (X)
40 #define VTH_max(X) (X)
41 #define VSH_max(X) (X)
42 #define VLTH_max(X) ((X)/5.0)
43 #define VSL_max(X) ((X)/5.0)
44 #define VTH_min(X) ((X)/2.0)
45 #define VSH_min(X) ((X)/2.0)
46 #define VLOGL_min(X) (0.0)
47 #define VLOGH_min(X) (0.0)
48 #define VLTH_min(X) (0.0)
49 #define VSL_min(X) (0.0)
50
51 /* The following formulas convert floating point voltage setting into */
52 /* an 8-bit DAC value based on a floating point DUT Vcc value */
53 /* X = input value to set (float), Y = DUT VCC (float) */
54 #define DAC_value_vthh(X,Y) (((Y) == 0.0)?0:(u_char)((((X)/(Y)-0.5)*2.0*256.0)))
55 #define DAC_value_vah(X,Y) DAC_value_vthh(X,Y)
56 #define DAC_value_vlth(X,Y) (((Y) == 0.0)?0:(u_char)((((X)/(Y)*5.0*256.0)))
57 #define DAC_value_vsl(X,Y) DAC_value_vlth(X,Y)
58 #define DAC_value_vlogl(X,Y) (((Y) == 0.0)?0:(u_char)((((X)/(Y)*256.0)))
59 #define DAC_value_vlogh(X,Y) DAC_value_vlogl(X,Y)
60
61 /* The following formulas convert an 8-bit DAC value into a floating point */
62 /* voltage setting based on a floating point DUT Vcc value */
63 /* X = DAC value to set (u_char), Y = DUT VCC (float) */
64 #define Float_value_vthh(X,Y) (((float)(X))/(2.0*256.0)+0.5) (Y)
65 #define Float_value_vah(X,Y) Float_value_vthh(X,Y)
66 #define Float_value_vlth(X,Y) (((float)(X))*5.0*256.0)
67 #define Float_value_vsl(X,Y) Float_value_vlth(X,Y)
68 #define Float_value_vlogl(X,Y) (((float)(X))*Y/(256.0))
69 #define Float_value_vlogh(X,Y) Float_value_vlogl(X,Y)
70
71 void
72 get_dutvcc(dutvcc)
73 float *dutvcc;
74 {
75     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
76     register u_char raw_dac_value;
77
78     /* Read DAC value twice (hardware requirement) */
79     raw_dac_value = pel->signal_adc;
80     raw_dac_value = pel->signal_adc;
81     *dutvcc = (float)(raw_dac_value) * ADC_LSB;
82 }
83
84 void
85 get_dutvcc(dutvcc)
86 float *dutvcc;
87 {
88     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
89     register u_char raw_dac_value;
90
91     /* Read DAC value twice (hardware requirement) */
92     raw_dac_value = pel->vcc_adc;
93     raw_dac_value = pel->vcc_adc;
94     *dutvcc = (float)(raw_dac_value) * DUTVCC_ADC_LSB;
95 }
96
97 int
98 pel_dac_adc()
99 {
100     int returncode = SUCCESS;
101     float dutvcc, dutanalog;
102
103     get_dutvcc(&dutvcc); /* Read the DUT Vcc line */
104
105     if(dab_type == NO_DAB)
106     {
107         if(dutvcc > DAC_ADC_ERROR)
108         {
109             (void)lm_error("Lane to Pel to DUT Vcc = %.3f exceeds no-DAB maximum.\n",
110                 current_lane + 'A', current_pel, dutvcc);
111             return(FAILURE);
112         }
113     }
114     else
115     {
116         if ((dutvcc > DUTVCC_max) || (dutvcc < DUTVCC_min))
117         {
118             (void)lm_error("DUT Vcc = %.3fV. Legal range is %.3fV to %.3fV.\n",
119                 dutvcc, DUTVCC_min, DUTVCC_max);
120         }
121     }
122 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_analog.c

DATE 5/23/89  
TIME 4:41:25 pm

PAGE #  
2/104

## SOURCE TEXT

```

121     return(FAILURE);
122 }
123
124 if(test_all_dacs(dutvcc, &returacode) != SUCCESS)
125     return(FAILURE);
126
127 if(dab_type == DIAG_DAB)
128 {
129     get_dutassalog(&dutassalog);
130     if((dutassalog > (1.0 + DUT_ANALOG_ERROR)) ||
131        (dutassalog < (1.0 - DUT_ANALOG_ERROR)))
132     {
133         (void)lm_error("DUTANALOG on Diagnostic DAB = %.3f. Expected \
134 1.000 V +/- %.3f V\n", dutassalog, DUT_ANALOG_ERROR);
135         return(FAILURE);
136     }
137 }
138 }
139 return(returacode);
140 }
141
142 int
143 test_a_dac(funcptr, charptr, dutvcc, returacode)
144 int (*funcptr)();
145 char *charptr;
146 float dutvcc;
147 int *returacode;
148 {
149     register int i;
150     register test;
151     static u_char value[9] =
152     {
153         0x00,
154         0x80,
155         0x01,
156         0x40,
157         0x02,
158         0x20,
159         0x04,
160         0x10,
161         0x08
162     };
163
164     if(dab_type == NO_DAB)
165     {
166         if(funcptr(0, 0.0) != SUCCESS)
167         {
168             *returacode = FAILURE;
169             if(lm_error("DAC test for %s (no DAB) fails.\n", charptr) != SUCCESS)
170                 return(FAILURE);
171         }
172     }
173     else
174     {
175         for(test = 0; test < 2; ++test) /* Test 0 => walking 0, else walking 1 */
176         {
177             for(i = 0; i < 9; ++i)
178             {
179                 if(funcptr(test == 0 ? value[i] : ~value[i], dutvcc) != SUCCESS)
180                 {
181                     *returacode = FAILURE;
182                     if(lm_error("DAC test for %s fails.\n", charptr) != SUCCESS)
183                         return(FAILURE);
184                 }
185             }
186         }
187     }
188     return(SUCCESS);
189 }
190
191 int
192 test_all_dacs(dutvcc, returacode)
193 float dutvcc;
194 int *returacode;
195 {
196     if(test_a_dac(set_vlogl_dac, "vlogl", dutvcc, returacode) != SUCCESS)
197         return(FAILURE);
198     if(test_a_dac(set_vlogh_dac, "vlogh", dutvcc, returacode) != SUCCESS)
199         return(FAILURE);
200     if(test_a_dac(set_vhth_dac, "vhth", dutvcc, returacode) != SUCCESS)
201         return(FAILURE);
202     if(test_a_dac(set_vsh_dac, "vsh", dutvcc, returacode) != SUCCESS)
203         return(FAILURE);
204     if(test_a_dac(set_vlth_dac, "vlth", dutvcc, returacode) != SUCCESS)
205         return(FAILURE);
206     if(test_a_dac(set_val_dac, "val", dutvcc, returacode) != SUCCESS)
207         return(FAILURE);
208     return(SUCCESS);
209 }
210
211 int
212 check_dac_value(value, min, max)
213 float value;
214 float min;
215 float max;
216 {
217     if ((value > max) || (value < min))
218     {
219         (void)lm_error("Desired DAC setting = %.3f out of range (%.3f - %.3f)\n",
220            value, min, max);
221         return(FAILURE);
222     }
223     return(SUCCESS);
224 }
225
226 int
227 set_vlogl_dac(dac_value, dutvcc)
228 u_char dac_value;
229 float dutvcc;
230 {
231     register PEL *pel = (PEL *) (pel_addr(current_lase, current_pel));
232     float value;
233
234     /* Compute the expected floating point value */
235     value = float_value_vlogl(dac_value, dutvcc);
236     if(set_dac_and_compare(&pel->vlogl_dac, &pel->vlogl_adc, dac_value, value) !=
237        SUCCESS)
238     {
239         (void)pel_error("Failure setting vlogl DAC.\n");
240         return(FAILURE);

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_analog.c	DATE 5/23/89	PAGE # 3/105
LINE #		SOURCE TEXT		
241		}		
242		return(SUCCESS);		
243		}		
244		int		
245		set_vlogl(value)		
246		float value;		
247		{		
248		float dutvcc;		
249		{		
250		get_dutvcc(&dutvcc); /* Read the DUT Vcc line */		
251		if(check_dac_value(value, VLOGL_min(dutvcc), VLOGL_max(dutvcc)) != SUCCESS)		
252		{		
253		(void)pel_error("Unable to set vlogl to desired value.\n");		
254		return(FAILURE);		
255		}		
256		return(set_vlogl_dac(DAC_value_vlogl(value, dutvcc), dutvcc));		
257		}		
258		int		
259		set_vlogh_dac(dac_value, dutvcc)		
260		u_char dac_value;		
261		float dutvcc;		
262		{		
263		register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));		
264		float value;		
265		{		
266		/* Compute the expected floating point value */		
267		value = float_value_vlogh(dac_value, dutvcc);		
268		if(set_dac_and_compare(&pel->vlogh_dac, &pel->vlogh_adc, dac_value, value) !=		
269		SUCCESS)		
270		{		
271		(void)pel_error("Failure setting vlogh DAC.\n");		
272		return(FAILURE);		
273		}		
274		return(SUCCESS);		
275		}		
276		int		
277		set_vlogh(value)		
278		float value;		
279		{		
280		float dutvcc;		
281		{		
282		get_dutvcc(&dutvcc); /* Read the DUT Vcc line */		
283		if(check_dac_value(value, VLOGH_min(dutvcc), VLOGH_max(dutvcc)) != SUCCESS)		
284		{		
285		(void)pel_error("Unable to set vlogh to desired value.\n");		
286		return(FAILURE);		
287		}		
288		return(set_vlogh_dac(DAC_value_vlogh(value, dutvcc), dutvcc));		
289		}		
290		int		
291		set_vlth_dac(dac_value, dutvcc)		
292		u_char dac_value;		
293		float dutvcc;		
294		{		
295		register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));		
296		float value;		
297		{		
298		/* Compute the expected floating point value */		
299		value = float_value_vlth(dac_value, dutvcc);		
300		if(set_dac_and_compare(&pel->vlth_dac, &pel->vlth_adc, dac_value, value) !=		
301		SUCCESS)		
302		{		
303		(void)pel_error("Failure setting vlth DAC.\n");		
304		return(FAILURE);		
305		}		
306		return(SUCCESS);		
307		}		
308		int		
309		set_vlth(value)		
310		float value;		
311		{		
312		float dutvcc;		
313		{		
314		get_dutvcc(&dutvcc); /* Read the DUT Vcc line */		
315		if(check_dac_value(value, VLTH_min(dutvcc), VLTH_max(dutvcc)) != SUCCESS)		
316		{		
317		(void)pel_error("Unable to set vlth to desired value.\n");		
318		return(FAILURE);		
319		}		
320		return(set_vlth_dac(DAC_value_vlth(value, dutvcc), dutvcc));		
321		}		
322		int		
323		set_val_dac(dac_value, dutvcc)		
324		u_char dac_value;		
325		float dutvcc;		
326		{		
327		register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));		
328		float value;		
329		{		
330		/* Compute the expected floating point value */		
331		value = float_value_val(dac_value, dutvcc);		
332		if(set_dac_and_compare(&pel->val_dac, &pel->val_adc, dac_value, value) !=		
333		SUCCESS)		
334		{		
335		(void)pel_error("Failure setting val DAC.\n");		
336		return(FAILURE);		
337		}		
338		return(SUCCESS);		
339		}		
340		int		
341		set_val(value)		
342		float value;		
343		{		
344		float dutvcc;		
345		{		
346		get_dutvcc(&dutvcc); /* Read the DUT Vcc line */		
347		if(check_dac_value(value, VSL_min(dutvcc), VSL_max(dutvcc)) != SUCCESS)		
348		{		
349		(void)pel_error("Unable to set val to desired value.\n");		
350		return(FAILURE);		
351		}		
352		return(set_val_dac(DAC_value_val(value, dutvcc), dutvcc));		
353		}		
354		int		
355		set_val(value)		
356		float value;		
357		{		
358		float dutvcc;		
359		{		
360		get_dutvcc(&dutvcc); /* Read the DUT Vcc line */		

[illegible]

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_analog.c

DATE

5/23/89

PAGE #

4/106

TIME

4:41:25 pm

LINE #

SOURCE TEXT

361

int

362

set\_vhth\_dac(dac\_value, dutvcc)

363

u\_char dac\_value;

364

float dutvcc;

365

{

366

register PEL \*pel = (PEL \*) (pel\_addr(current\_lase, current\_pel));

367

float value;

368

/\* Compute the expected floating point value \*/

369

value = Float\_value\_vhth(dac\_value, dutvcc);

370

if(set\_dac\_and\_compare(&pel->vhth\_dac, &pel->vhth\_adc, dac\_value, value) !=

371

SUCCESS)

372

{

373

(void)pel\_error("Failure setting vhth DAC.\n");

374

return(FAILURE);

375

}

376

return(SUCCESS);

377

}

378

int

379

set\_vhth(value)

380

float value;

381

{

382

float dutvcc;

383

get\_dutvcc(&dutvcc);

384

/\* Read the DUT Vcc line \*/

385

if(check\_dac\_value(value, VTH\_min(dutvcc), VTH\_max(dutvcc)) != SUCCESS)

386

{

387

(void)pel\_error("Unable to set vhth to desired value.\n");

388

return(FAILURE);

389

}

390

return(set\_vhth\_dac(DAC\_value\_vhth(value, dutvcc), dutvcc));

391

}

392

int

393

set\_vah\_dac(dac\_value, dutvcc)

394

u\_char dac\_value;

395

float dutvcc;

396

{

397

register PEL \*pel = (PEL \*) (pel\_addr(current\_lase, current\_pel));

398

float value;

399

u\_char adc\_value;

400

/\* Compute the expected floating point value \*/

401

if(dac\_type == NO\_DAB) /\* Special case during DAC/ADC test w/o DAB \*/

402

{

403

/\* Read ADC twice (hardware requirement) \*/

404

adc\_value = pel->vah\_adc;

405

/\* Delay 1s; \*/

406

value = ((float)(adc\_value - pel->vah\_adc)) \* ADC\_LSB;

407

/\* Compare result against vah without DAB upper bound \*/

408

if(value > (VSH\_NO\_DAB + DAC\_ADC\_ERROR))

409

{

410

(void)lm\_error("DAC set to 11.3f (%02X). ADC measured 11.3f (%02X).\n",

411

VSH\_NO\_DAB, dac\_value, value, adc\_value);

412

return(FAILURE);

413

}

414

return(SUCCESS);

415

}

416

else

417

value = Float\_value\_vah(dac\_value, dutvcc);

418

if(set\_dac\_and\_compare(&pel->vah\_dac, &pel->vah\_adc, dac\_value, value) !=

419

SUCCESS)

420

{

421

(void)pel\_error("Failure setting vah DAC.\n");

422

return(FAILURE);

423

}

424

return(SUCCESS);

425

}

426

int

427

set\_vah(value)

428

float value;

429

{

430

float dutvcc;

431

get\_dutvcc(&dutvcc);

432

/\* Read the DUT Vcc line \*/

433

if(check\_dac\_value(value, VSH\_min(dutvcc), VSH\_max(dutvcc)) != SUCCESS)

434

{

435

(void)pel\_error("Unable to set vah to desired value.\n");

436

return(FAILURE);

437

}

438

return(set\_vah\_dac(DAC\_value\_vah(value, dutvcc), dutvcc));

439

}

440

int

441

read\_adc\_and\_compare(adcptr, dac\_value, expected\_value)

442

u\_char \*adcptr;

443

u\_char dac\_value;

444

float expected\_value;

445

{

446

float measured\_value, measured\_value\_II;

447

u\_char adc\_value;

448

/\* Read ADC twice (hardware requirement) \*/

449

adc\_value = \*adcptr;

450

/\* Delay 1s; \*/

451

measured\_value = ((float)(adc\_value - \*adcptr)) \* ADC\_LSB;

452

/\* Delay 1s; \*/

453

measured\_value\_II = ((float)(adc\_value - \*adcptr)) \* ADC\_LSB;

454

/\* Compare values \*/

455

if ((measured\_value > (expected\_value + DAC\_ADC\_ERROR)) ||

456

(measured\_value < (expected\_value - DAC\_ADC\_ERROR)))

457

{

458

(void)lm\_error("DAC set to 11.3f (%02X). ADC first measurement 11.3f (%02X).\n",

459

expected\_value, dac\_value, measured\_value, adc\_value);

460

(void)lm\_error("DAC set to 11.3f (%02X). ADC second measurement 11.3f (%02X).\n",

461

expected\_value, dac\_value, measured\_value\_II, adc\_value);

462

return(FAILURE);

463

}

464

return(SUCCESS);

465

}

466

int

467

set\_dac\_and\_compare(dacptr, adcptr, dac\_value, expected\_value)

468

u\_char \*dacptr;

469

u\_char \*adcptr;

470

u\_char dac\_value;

471

float expected\_value;

472

{

473

float measured\_value;

474

/\* Read ADC twice (hardware requirement) \*/

475

measured\_value = \*adcptr;

476

/\* Delay 1s; \*/

477

/\* Delay 1s; \*/

478

/\* Compare values \*/

479

if ((measured\_value > (expected\_value + DAC\_ADC\_ERROR)) ||

480

(measured\_value < (expected\_value - DAC\_ADC\_ERROR)))

[illegible]

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_analog.c	DATE 5/23/89	PAGE # 7/109
TIME 4:41:25 pm				
LINE #	SOURCE TEXT			
721	(void)lm_message("%5.3f ", result);			
722	result = pel->val_adc * ADC_LSB;			
723	result = pel->val_adc * ADC_LSB;			
724	(void)lm_message("%5.3f ", result);			
725	(void)lm_message("%5.3f ", Float_value_vlth(dac_value,dutvcc));			
726				
727	result = pel->vlth_adc * ADC_LSB;			
728	result = pel->vlth_adc * ADC_LSB;			
729	(void)lm_message("%5.3f ", result);			
730	result = pel->vlth_adc * ADC_LSB;			
731	result = pel->vlth_adc * ADC_LSB;			
732	(void)lm_message("%5.3f ", result);			
733	(void)lm_message("%5.3f ", Float_value_vlth(dac_value,dutvcc));			
734				
735	result = pel->vth_adc * ADC_LSB;			
736	result = pel->vth_adc * ADC_LSB;			
737	(void)lm_message("%5.3f ", result);			
738	result = pel->vth_adc * ADC_LSB;			
739	result = pel->vth_adc * ADC_LSB;			
740	(void)lm_message("%5.3f ", result);			
741	(void)lm_message("%5.3f ", Float_value_vth(dac_value,dutvcc));			
742				
743	(void)lm_message("\n");			
744				
745				



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM	DATE	PAGE #
		diags/pel_crc.c	5/23/89	1/110
			TIME 4:41:26 pm	
LINE #	SOURCE TEXT			
1	/* SCCS_ID: pel_crc.c rev 1.1, 4/24/89 at 07:49:50 */			
2				
3	/*			
4	*****			
5	*****			
6	*****			
7	*****			
8	*****			
9	*****			
10	*****			
11	*****			
12	*****			
13	*****			
14	*****			
15	*****			
16	*****			
17	*****			
18	*****			
19	*****			
20	*****			
21	*****			
22	*****			
23	*****			
24	*****			
25	*****			
26	*****			
27	*****			
28	*****			
29	*****			
30	*****			
31	*****			
32	*****			
33	*****			
34	*****			
35	*****			
36	*****			
37	*****			
38	*****			
39	*****			
40	*****			
41	*****			
42	*****			
43	*****			
44	*****			
45	*****			
46	*****			
47	*****			
48	*****			
49	*****			
50	*****			
51	*****			
52	*****			
53	*****			
54	*****			
55	*****			
56	*****			
57	*****			
58	*****			
59	*****			
60	*****			
61	*****			
62	*****			
63	*****			
64	*****			
65	*****			
66	*****			
67	*****			
68	*****			
69	*****			
70	*****			
71	*****			
72	*****			
73	*****			
74	*****			
75	*****			
76	*****			
77	*****			
78	*****			
79	*****			
80	*****			
81	*****			
82	*****			
83	*****			
84	*****			
85	*****			
86	*****			
87	*****			
88	*****			
89	*****			
90	*****			
91	*****			
92	*****			
93	*****			
94	*****			
95	*****			
96	*****			
97	*****			
98	*****			
99	*****			
100	*****			
101	*****			
102	*****			
103	*****			
104	*****			
105	*****			
106	*****			
107	*****			
108	*****			
109	*****			
110	*****			
111	*****			
112	*****			
113	*****			
114	*****			
115	*****			
116	*****			
117	*****			
118	*****			
119	*****			
120	*****			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_crc.c

DATE 5/23/89  
TIME 4:41:26 pm

PAGE #  
2/111

## SOURCE TEXT

```

121 (void)pel_error("Unable to initialize PEL/DAB for CRC test.\n");
122 return(FAILURE);
123 }
124 return(pel_set_diag_dab_period(PAC_MIN_PERIOD * 1000));
125 }
126
127 int
128 pel_set_diag_dab_period(clock_period)
129 int clock_period; /* is picoseconds */
130 {
131     register int edge;
132     char buffer[80]; /* Default edges for Diagnostic Adapter */
133     long edgetime[6]; /* edge times are close */
134     if(clock_period < 200000) /* edge times are close */
135     {
136         edgetime[0] = 10000;
137         edgetime[1] = 10000;
138         edgetime[2] = 10000;
139         edgetime[3] = 10000;
140         edgetime[4] = 10000;
141         edgetime[5] = 0;
142     }
143     else /* edge times are spread out */
144     {
145         edgetime[0] = 500000;
146         edgetime[1] = 500000;
147         edgetime[2] = 500000;
148         edgetime[3] = 1000000;
149         edgetime[4] = 500000;
150         edgetime[5] = 0;
151     }
152     if(pel_debug & 2)
153     {
154         diag_get_log((long)clock_period, "Period", 400001, 8000000001);
155         for(edge = 0; edge < 6; ++edge)
156         {
157             sprintf(buffer, "Edge %d", edge);
158             diag_get_log((edgetime[edge]), buffer, 01, (long)clock_period);
159         }
160     }
161     if(tmg_set_timing((long)clock_period, edgetime, 01, 1000001, 25000001)
162        != SUCCESS)
163     {
164         (void)lm_error("Unable to set edges (tdps clock period) for CRC test.\n",
165                      clock_period);
166         return(FAILURE);
167     }
168     return(SUCCESS);
169 }
170
171 int
172 pel_crc_test_128K(first_pattern) /* Must run pel_crc_setup before running */
173 int first_pattern;
174 {
175     register int byte;
176     int returncode = SUCCESS;
177     for(byte = 0; byte < 2; ++byte)
178     {
179         if(pel_play_crc_sequence(first_pattern, PATTERNS_IN_128K, byte) != SUCCESS)
180         {
181             returncode = FAILURE;
182             if(lm_error("CRC sequence fails while testing %s byte.\n",
183                      byte > "high" : "low") != SUCCESS)
184                 return(FAILURE);
185         }
186     }
187     return(returncode);
188 }
189
190 /* PEL CRC Test:
191 */
192 pel_crc()
193 {
194     if(diag_clear_errors() != SUCCESS)
195         return(FAILURE);
196     if(pel_crc_setup() != SUCCESS)
197     {
198         (void)pel_error("Unable to perform Pin Electronics CRC test.\n");
199         return(FAILURE);
200     }
201     return(pel_crc_test_128K(0));
202 }
203
204 int
205 pel_play_crc_sequence(first_pattern, num_patterns, byte)
206 int first_pattern;
207 int num_patterns;
208 int byte;
209 {
210     register PEL *pel = (PEL *) (pel_addr(current_1ane, current_pel));
211     register int pattern_count;
212     register int time_out;
213     register int crc_patterns = num_patterns - BLOCK_SIZE;
214     register int shift;
215     register int chip;
216     register int pass;
217     int i;
218     int magic_0;
219     int crc_result;
220     int returncode = SUCCESS;
221     char amble_buffer[256];
222
223     static u_long right_crc_result[2][10] =
224     {
225         {
226             0x1021D,
227             0x0B46E,
228             0x18A93,
229             0x10AD3,
230             0x04852,
231             0x0D26A,
232         }
233     }
234     /* Results for pass 0 */

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_crc.c

DATE

5/23/89

PAGE #

TIME

4:41:26 pm

3/112

```

LINE # SOURCE TEXT
241 0x01C0C,
242 0x1948D,
243 0x00977,
244 0x09E4F,
245 },
246 {
247 0x1C106,
248 0x1CBF4,
249 0x1BFD7,
250 0x11323,
251 0x15846,
252 /* Results for pass 1 */
253 0x01D45,
254 0x19357,
255 0x03CE9,
256 0x00ACE,
257 },
258 },
259
260 if((time_out = pac_play()) == 0)
261 {
262 (void)pel_error("Unable to prepare for play.\n");
263 return(FAILURE);
264 }
265
266 for(pass = 0; pass < 2; ++pass)
267 {
268 if(pel_fill_crc(first_pattern + (pass ? BLOCK_SIZE : 0), crc_patterns,
269 current_pel_byte, DOAB_TEST_HDATA : DOAB_TEST_LODATA, byte,
270 (int)(SEED << pass)) != SUCCESS)
271 {
272 (void)pel_error("Unable to fill pattern memory with CRC data.\n");
273 return(FAILURE);
274 }
275
276 if(pel_load_pattern_string(CRC_SHIFT_PATTERN, first_pattern +
277 (pass ? 0 : crc_patterns), STOP_MODE) == 0)
278 {
279 (void)pel_error("Unable to load shift pattern string in CRC test.\n");
280 return(FAILURE);
281 }
282
283 (void)lm_message("Start pattern %i. Pass %d, to Byte: Playing patterns",
284 first_pattern, pass, byte ? "High" : "Low");
285
286 for(chip = 0; chip < 5; ++chip)
287 {
288 lm_message("."); /* Walking dots while messages are on */
289 (void)generate_preamble(amble_buffer, chip, byte);
290 if((build_pattern_data(amble_buffer, first_pattern +
291 (pass ? BLOCK_SIZE : 0)) == 0)
292 {
293 (void)pel_error("Unable to build preamble pattern data.\n");
294 return(FAILURE);
295 }
296
297 pattern_count = generate_postamble(amble_buffer, byte);
298 if((build_pattern_data(amble_buffer, first_pattern +
299 crc_patterns - pattern_count + (pass ? BLOCK_SIZE : 0)) == 0)
300 {
301 (void)pel_error("Unable to build postamble pattern data.\n");
302 return(FAILURE);
303 }
304
305 if((build_pattern_control(first_pattern + (pass ? BLOCK_SIZE : 0),
306 crc_patterns, STOP_MODE) != SUCCESS)
307 {
308 (void)pel_error("Could not build CRC pattern control.\n");
309 return(FAILURE);
310 }
311 if(pac_play(time_out) != SUCCESS)
312 {
313 (void)pel_error("Pattern play failed during CRC tests.\n");
314 return(FAILURE);
315 }
316 /* Read the CRC result from magic chip 0 by playing a shift */
317 /* sequence twice and concatenating the results to form a 17 bit */
318 /* checksum */
319
320 for(crc_result = shift = 0; shift < 3; ++shift)
321 {
322 if(pel_debug & 4) /* Read raw data out of all five magic chips */
323 for(i = 0; i < 5; ++i)
324 {
325 (void)lm_message("Pass %d Byte %d Chip %d Shift %d Magic %d: %04X\n",
326 pass, byte, chip, shift, i, pel->magic_chip[i].reg[11]);
327 }
328
329 /* Make sure control word byte is 0x04 */
330 if(((magic_0 = pel->magic_chip[0].reg[11]) & 0xff00) != 0x0400)
331 {
332 returncode = FAILURE;
333 if(lm_error("Magic chip CRC control byte not equal to 04 \
334 (actual %02X).\n", (magic_0 >> 8) & 0xff) != SUCCESS)
335 return(FAILURE);
336 }
337
338 crc_result |= ((magic_0 & 0xff) < (8 * shift));
339 if(shift < 2) /* Shift CRC data by playing the shift pattern */
340 {
341 pac_set_first_block(Lane_code(current_lane), (first_pattern +
342 (pass ? 0 : crc_patterns)) / BLOCK_SIZE);
343 if(pac_play(time_out) != SUCCESS)
344 {
345 (void)pel_error("Pattern play failed trying to shift CRC data.\n");
346 return(FAILURE);
347 }
348 }
349 }
350
351 crc_result ^= 0x1ffff;
352
353 /* Compare the actual CRC result with the expected result */
354 if(crc_result != right_crc_result[pass][5 * byte + chip])
355 {
356 returncode = FAILURE;
357 if(lm_error("Pass %d, Byte %d, Chip %d, CRC = %05x. Expected %05x.\n",
358 pass, byte, chip, crc_result, right_crc_result[pass][5 * byte + chip])
359 != SUCCESS)
360 return(FAILURE);

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_crc.c	DATE 5/23/89	PAGE # 4/113
LINE #		SOURCE TEXT	TIME 4:41:26 pm	
361		if(pel_debug & 1)		
362		msg_play_tty_key();		
363		}		
364		(void)lm_message("done.\n");		
365		}		
366		return(returncode);		
367		}		
368		return(returncode);		
369		}		
370				
371		int pel_fill_crc(first_patterns, patterns, pel_no, control, dbyte, seed)		
372		/*		
373		INPUT: first_patterns = first patterns number to fill (0's counting)		
374		patterns = number of patterns to fill		
375		pel_no = four bit PEL number code		
376		control = eight bit control word		
377		dbyte : 0 => data in low byte, 1 => data in high byte		
378		seed = random number seed		
379		OUTPUT: return code = SUCCESS or FAILURE		
380		DESCRIPTION: fills pattern memory with pseudorandom		
381		data as well as control bits. Used during PAC/PEL CRC checksum		
382		diagnostics.		
383		*/		
384				
385		int		
386		pel_fill_crc(first_patterns, patterns, pel_no, control, dbyte, seed)		
387		int first_patterns;		
388		int patterns;		
389		int pel_no;		
390		int control;		
391		int dbyte;		
392		int seed;		
393		{		
394		register int dbyte = (dbyte == 0); /* dbyte is opposite dbyte */		
395		register u_long *memptr;		
396		register u_long i;		
397		register u_long fill_mask;		
398		register u_long fill_data;		
399		register u_long random_number;		
400		u_long base_address;		
401		u_long bank_offset;		
402		int bank;		
403		{		
404		if(pel_fill_check(first_patterns, patterns) != SUCCESS)		
405		return(FAILURE);		
406				
407		base_address = pel[current_lase].lase_offset + 4 * first_patterns;		
408		random_number = seed;		
409				
410		/* Compute fill_mask and fill_data for banks 0 and 1 (data only) */		
411		fill_mask = 0x7f << ((8 * dbyte));		
412		fill_mask  = fill_mask << 16;		
413		fill_mask = ~fill_mask;		
414		fill_data = ((u_long)control & 0x7f) << ((8 * dbyte));		
415		fill_data  = fill_data << 16;		
416		for(bank = 0, bank_offset = 0, bank < 3; ++bank)		
417		{		
418		if(bank == 2) /* modify fill_mask and fill_data */		
419		{		
420		fill_mask = (fill_mask   0xffff) & ~0x7fff;		
421		fill_data = (fill_data & 0xffff)   ((u_long)pel_no & 0xf)   (0x7 << 4);		
422		}		
423		memptr = (u_long *) (base_address + bank_offset);		
424		if(patterns > 1)		
425		{		
426		i = patterns - 1; /* do loop for speed */		
427		do		
428		{		
429		random_number = Pel_get_random(random_number);		
430		*memptr++ = (random_number & fill_mask)   fill_data;		
431		} while (--i);		
432		}		
433		*memptr++ = fill_data;		
434		bank_offset += PAT_BANK_SIZE;		
435		}		
436		return(SUCCESS);		
437		}		
438				
439				
440		/*		
441		int pel_build_crc_check(pel_no, check_edge, seq_length)		
442		/*		
443		INPUT: pel_no = four bit PEL number code		
444		check_edge = branch code (BRANCH_FALLING or BRANCH_RISING)		
445		seq_length = 0, 1, or 512		
446		OUTPUT: returns number of blocks written (0 => FAILURE)		
447		DESCRIPTION: Builds feedback sequence in first several blocks		
448		of pattern memory.		
449		*/		
450		int		
451		pel_build_crc_check(pel_no, check_edge, seq_length)		
452		int pel_no;		
453		int check_edge;		
454		int seq_length;		
455		{		
456		PAT_WORD pattern_word, *patptr = &pattern_word;		
457		u_long *memptr;		
458		int pattern_no;		
459		int word;		
460		int i;		
461		int blocks_filled;		
462		/* Fill first two blocks of pattern memory with 0's in the lower byte */		
463		/* of each magic chip pattern word and diag DAB control in the other */		
464		/* (the control word does not contain a clock bit) */		
465		if(pel_fill_crc(0, 2 * BLOCK_SIZE, pel_no, DIAG_TEST_LOADNO_CLK, 0, 0)		
466		!= SUCCESS)		
467		{		
468		(void)pel_error("Unable to fill first two blocks in pattern memory.\n");		
469		return(FAILURE);		
470		}		
471		memptr = (u_long *) Pattern_to_address(current_lase, 2, 0);		
472		switch(seq_length)		
473		{		
474		case 0: /* Place branch (no clock command) every 5 locations */		
475		for(i = 0; i < (BLOCK_SIZE / 4); ++i)		
476		{		
477		*memptr++ = 7;		
478		*memptr++  = check_edge;		
479		}		
480		break;		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_crc.c	DATE 5/23/89 TIME 4:41:26 pm	PAGE # 5/114
LINE #	SOURCE TEXT			
481	/* Place clock command and branch every 8 locations */			
482	pattern_no = 0;			
483	while(pattern_no < (2 * BLOCK_SIZE))			
484	{			
485	(void)pac_read_pattern(pattern_no, patptr);			
486	for(word = 0; word < 5; ++word)			
487	patptr->pattern_data[word]  = DDAB_TEST_LODATA << 8;			
488	(void)pac_write_pattern(pattern_no, patptr);			
489	numptr += 7;			
490	numptr += feedback_edge;			
491	pattern_no += 8;			
492	}			
493	break;			
494	/* Place one clock command and branch near end of block */			
495	pattern_no = (2 * BLOCK_SIZE) - 8;			
496	(void)pac_read_pattern(pattern_no, patptr);			
497	for(word = 0; word < 5; ++word)			
498	patptr->pattern_data[word]  = DDAB_TEST_LODATA << 8;			
499	(void)pac_write_pattern(pattern_no, patptr);			
500	numptr += (2 * BLOCK_SIZE) - 1;			
501	numptr += feedback_edge;			
502	break;			
503	default:			
504	(void)pel_error("Software failure - feedback sequence length invalid.\n");			
505	return(0);			
506	}			
507	/* Find out size of PPM 80 and copy first two blocks appropriate number */			
508	/* of times */			
509	switch(pac_get_ppm_size(current_lane, 0))			
510	{			
511	case PATTERNS_IN_128K:			
512	return(2);			
513	case PATTERNS_IN_512K:			
514	blocks_filled = 4;			
515	break;			
516	case PATTERNS_IN_3M:			
517	blocks_filled = 8;			
518	break;			
519	default:			
520	(void)pel_error("Pattern memory size not valid.\n");			
521	return(0);			
522	}			
523	/* Now copy the first two blocks (blocks_filled/2); number of times */			
524	for(i = 0; i <= (blocks_filled >> 3); ++i)			
525	{			
526	if(pel_copy_pat_mem(0, (2 * BLOCK_SIZE) << 1, (2 * BLOCK_SIZE) << 1)			
527	!= SUCCESS)			
528	{			
529	(void)pel_error("Unable to copy pattern memory.\n");			
530	return(0);			
531	}			
532	}			
533	return(blocks_filled);			
534	}			
535	int			
536	pel_copy_pat_mem(from_pattern, to_pattern, num_patterns)			
537	{			
538	register int from_pattern;			
539	register int to_pattern;			
540	register int num_patterns;			
541	{			
542	PAT_WORD pattern_word, *patptr = spattern_word;			
543	register int pattern_no;			
544	{			
545	if((pac_fill_check(from_pattern, num_patterns) != SUCCESS)			
546	(pac_fill_check(to_pattern, num_patterns) != SUCCESS))			
547	return(FAILURE);			
548	{			
549	for(pattern_no = 0; pattern_no < num_patterns; ++pattern_no)			
550	{			
551	(void)pac_read_pattern(pattern_no + from_pattern, patptr);			
552	(void)pac_write_pattern(pattern_no + to_pattern, patptr);			
553	}			
554	return(SUCCESS);			
555	}			
556	pel_feedback_test()			
557	{			
558	PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));			
559	int returncode = SUCCESS;			
560	int i;			
561	int preamble_length;			
562	int postamble_length;			
563	char amble_buffer[256];			
564	int num_blocks;			
565	int patterns;			
566	int added_patterns;			
567	int none;			
568	int ptr_offset;			
569	int crc_result;			
570	long *numptr;			
571	int dummy;			
572	{			
573	if(diag_clear_errors() != SUCCESS)			
574	return(FAILURE);			
575	{			
576	/* Initialize the PEL and the Diagnostic DAB */			
577	if(pel_crc_set() != SUCCESS)			
578	{			
579	(void)pel_error("Unable to perform feedback test.\n");			
580	return(FAILURE);			
581	}			
582	{			
583	/* First place a feedback sequence starting at block 0 in pattern memory */			
584	if((num_blocks = pel_build_crc_fbseq(current_pel, BRANCH_FALLING, 0)) == 0)			
585	return(FAILURE);			
586	{			
587	/* Next, place the CRC preamble and postamble in pattern memory */			
588	/* Set the memory pointer to the beginning of pattern memory, bank 2 */			
589	numptr = (long *) pattern_to_address(current_lane, 2, 0);			
590	/* First, add the preamble */			
591	preamble_length = generate_preamble(amble_buffer, 0, 0);			
592	if(build_pattern_data(amble_buffer, num_blocks + BLOCK_SIZE) == 0)			
593	{			
594	(void)pel_error("Unable to build preamble pattern data.\n");			
595	return(FAILURE);			
596	}			
597	{			
598	/* Now we add patterns to write a one into the CRC circuit */			
599	if(build_pattern_data(FRAME_LOAD, (num_blocks + BLOCK_SIZE) +			
600	)			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_crc.c	DATE 5/23/89	PAGE # 6/115
LINE #		SOURCE TEXT		
601		preamble_length == 0)		
602		{		
603		(void)pel_error("Unable to add feedback load patterns to preamble.\n");		
604		return(FAILURE);		
605		}		
606		preamble_length += FBANK_LOAD_SIZE;		
607		/* Add at least 8 NOP patterns to end of preamble */		
608		added_patterns = 8;		
609		/* Add a branch always command to the preamble block */		
610		ptr_offset = preamble_length + added_patterns - BRANCH_LATENCY;		
611		if((sops = 1 - ptr_offset) > 0)		
612		{		
613		added_patterns += sops;		
614		ptr_offset += sops;		
615		}		
616		if((ptr_offset & 1) == 0)		
617		{		
618		++added_patterns;		
619		++ptr_offset;		
620		}		
621		/* Add NOP patterns to end of preamble */		
622		if(pel_fill_crc((num_blocks * BLOCK_SIZE) + preamble_length,		
623		added_patterns, current_pel, DDAB_TEST_LODATA_NO_CLK, 0, 0) != SUCCESS)		
624		{		
625		(void)pel_error("Unable to add NOP patterns to preamble.\n");		
626		return(FAILURE);		
627		}		
628		preamble_length += added_patterns;		
629		*(numptr + (num_blocks * BLOCK_SIZE) + ptr_offset)  = BRANCH_ALWAYS;		
630		}		
631		/* Now, add the postamble */		
632		postamble_length = generate_postamble(amble_buffer, 0);		
633		added_patterns = ((ptr_offset + postamble_length - STOP_LATENCY) < 0) ?		
634		-ptr_offset : 0;		
635		if(added_patterns > 0) /* Add NOP patterns to beginning of postamble */		
636		{		
637		if(pel_fill_crc((num_blocks + 1) * BLOCK_SIZE, added_patterns, current_pel,		
638		DDAB_TEST_LODATA_NO_CLK, 0, 0) != SUCCESS)		
639		{		
640		(void)pel_error("Unable to add NOP patterns to postamble.\n");		
641		return(FAILURE);		
642		}		
643		}		
644		if(build_patterns_data(amble_buffer, ((num_blocks + 1) * BLOCK_SIZE) +		
645		added_patterns) == 0)		
646		{		
647		(void)pel_error("Unable to build postamble pattern data.\n");		
648		return(FAILURE);		
649		}		
650		postamble_length += added_patterns;		
651		*(numptr + ((num_blocks + 1) * BLOCK_SIZE) + ptr_offset + added_patterns)  =		
652		STOP;		
653		}		
654		/* Load the link table */		
655		numptr = (long *)(&pac[current_lane].lane_offset + LINK_OFFSET);		
656		*numptr = num_blocks + 1;		
657		numptr += num_blocks;		
658		*numptr = 0;		
659		}		
660		pac_clock_speed(current_lane, PAC_MIN_PERIOD);		
661		for(i = 0; i < 2; ++i)		
662		{		
663		/* Check for backplane errors */		
664		if(report_by_error() != 0)		
665		{		
666		(void)pel_error("Backplane errors - can't initiate play.\n");		
667		return(FAILURE);		
668		}		
669		pac_set_first_block(Lane_code(current_lane), num_blocks);		
670		/* Play the pattern (should time out) */		
671		if(tmg_initiate_play() != SUCCESS)		
672		{		
673		pac_play_cleanup();		
674		return(FAILURE);		
675		}		
676		lm_delay(1); /* Wait 5ms (plenty of time at 15 MHz) */		
677		if(bp_mode() != PLAY_MODE) /* Should still be in play mode */		
678		{		
679		returncode = FAILURE;		
680		if(lm_error("Pattern play completed - expected time out.\n") != SUCCESS)		
681		return(FAILURE);		
682		}		
683		else		
684		{		
685		if(pac_abort_play() != SUCCESS)		
686		{		
687		pac_play_cleanup();		
688		return(FAILURE);		
689		}		
690		}		
691		if(i == 1)		
692		break;		
693		/* Now place a rising feedback sequence in pattern memory */		
694		if(pel_build_crc_fbck(current_pel, BRANCH_RISING, 0) != num_blocks)		
695		return(FAILURE);		
696		}		
697		for(i = 0; i < 2; ++i)		
698		{		
699		/* Place an 8-cycle feedback sequence in pattern memory */		
700		if(pel_build_crc_fbck(current_pel, i ? BRANCH_RISING : BRANCH_FALLING, 8)		
701		!= num_blocks)		
702		return(FAILURE);		
703		pac_set_first_block(Lane_code(current_lane), num_blocks);		
704		if(pac_play(TIMEOUT) != SUCCESS)		
705		{		
706		(void)pel_error("Pattern play failed during 8-cycle feedback.\n");		
707		return(FAILURE);		
708		}		
709		/* Read CRC result and compare with expected */		
710		crc_result = pel->magic_chip[0].reg[11] & 0xff;		
711		if(crc_result != (i ? RISING_8_CRC : FALLING_8_CRC))		
712		{		
713		returncode = FAILURE;		
714		if(lm_error("CRC result does not match. Expected %02X. Actual %02X.\n",		
715		i ? RISING_8_CRC : FALLING_8_CRC, crc_result) != SUCCESS)		
716		return(FAILURE);		
717		}		
718		}		
719		if(pel_set_diag_dab_period(PAC_MAX_PERIOD * 1000) != SUCCESS)		
720		{		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_crc.c

DATE 5/23/89  
TIME 4:41:26 pm

PAGE #  
7/116

```

SOURCE TEXT
LINE #
721 {
722 (void)pel_error("Unable to set clock to min frequency in feedback test.\n");
723 return(FAILURE);
724 }
725 Pac_clock_speed(current_lane, PAC_MAX_PERIOD);
726
727 for(i = 0; i < 2; ++i)
728 {
729 /* Place a 512-cycle feedback sequence in pattern memory */
730 if(pel_build_crc_check(current_pel, 1 ? BRANCH_RISING : BRANCH_FALLING, 512)
731 != sum_blocks)
732 return(FAILURE);
733 patterns = preamble_length + postamble_length + BRANCH_LATENCY - 1 +
734 (2 * BLOCK_SIZE) * (1 ? 16 : 17);
735 pac_set_first_block(lane_code(current_lane), sum_blocks);
736 if(pac_timed_play(PAC_MAX_PERIOD, patterns, &dummy) != SUCCESS)
737 {
738 returncode = FAILURE;
739 if(lane_error("Timed pattern play failed during 512-cycle feedback.\n")
740 != SUCCESS)
741 return(FAILURE);
742 }
743 /* Read CRC result and compare with expected */
744 crc_result = pel_magic_chip[0].reg[11] & 0x1f;
745 if(crc_result != (1 ? RISING_512_CRC : FALLING_512_CRC))
746 {
747 returncode = FAILURE;
748 if(lane_error("CRC result does not match. Expected %02X. Actual %02X.\n",
749 1 ? RISING_512_CRC : FALLING_512_CRC, crc_result) != SUCCESS)
750 return(FAILURE);
751 }
752 }
753
754 return(returncode);
755 }
756
757 set_user_bit(patterns_no, value)
758 {
759 PAT_WORD pattern_word;
760
761 (void)pac_read_patterns(patterns_no, &pattern_word);
762 pattern_word.pattern_mask = value;
763 (void)pac_write_patterns(patterns_no, &pattern_word);
764 }
765
766 #define IMUSELED_MASK 0x0000
767 #define KEEPALIVE_MASK 0x0000
768 #define CLK10MHz2_MASK 0x0000
769 #define CLK10MHz2_MASK 0x0000
770 #define USER_MASK 0x0000
771 #define KEEPALIVE_PATTERN "2222227"
772 #define MAX_CLK10MHz2_ATTEMPTS 100
773 #define MAX_CLK10MHz2_ATTEMPTS 50
774
775 pel_keepalive_test()
776 {
777 static long edgetime[5] = {
778 100001, 100001, 100001, 100001, 100001
779 };
780 static long period = 400001;
781 int pac_replay();
782 register PEL *pel;
783 u_short patterns[5];
784 int attempts, differences, found_a_0, found_a_1;
785 int errors = 0;
786
787 if(diag_clear_errors() != SUCCESS)
788 return(FAILURE);
789
790 if (set_vlog1(0.8) == FAILURE) return(FAILURE);
791 if (set_vlog2(2.0) == FAILURE) return(FAILURE);
792 if (set_vhth(0.3) == FAILURE) return(FAILURE);
793 if (set_val(0.4) == FAILURE) return(FAILURE);
794 if (set_vah(3.0) == FAILURE) return(FAILURE);
795 if (set_vhth(4.4) == FAILURE) return(FAILURE);
796
797 if (pel_lane_init() != SUCCESS) { /* set up pac/tmg/pel/dab */
798 (void)lan_error("Unable to initialize lane to\n", current_lane + "A");
799 return(FAILURE);
800 }
801
802 if(pel_diag_dab_test_init(PUBLIC DAB) != SUCCESS) {
803 (void)pel_error("Unable to initialize PEL/DAB for keepalive/trigger bit test.\n");
804 return(FAILURE);
805 }
806
807 if (tmg_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) {
808 return FAILURE;
809 }
810
811 /*
812 1. Verify that nothing is driving bus(16..1)
813 */
814 if (pel_play_pattern_string("2222227", 0) != SUCCESS) {
815 (void)pel_error("Unable to play keepalive/trigger bit test patterns.\n");
816 return(FAILURE);
817 }
818
819 patterns[4] = patterns[3] = patterns[2] = patterns[1] = patterns[0] = 0x0000;
820 if (compare_magic_chips(11, patterns) != SUCCESS) { /* VALUE SAMPLE */
821 (void)pel_error("Failed buffer 112 test.\n");
822 errors++;
823 }
824
825 /*
826 2. Verify USER=1, IMUSE LED=0, KEEPALIVE=1, CLK 10MHz2=1 and CLK 1MHz2=1
827 */
828 pel->car.bit.eeprom_sel = 1;
829 pel->car.bit.in_use_led = 1;
830 if (pel_load_patterns_string(KEEPALIVE_PATTERN, 0, STOP_MODE) == FAILURE) {
831 (void)pel_error("Unable to load keepalive/trigger bit test patterns.\n");
832 return(FAILURE);
833 }
834
835 (void)set_user_bit(styles(KEEPALIVE_PATTERN)/2 - 1, 1);
836 if(diag_play() != SUCCESS) {

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_crc.c	DATE 5/23/89	PAGE # 8/117
LINE #	SOURCE TEXT			
841	<pre> (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return(Failure); } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; INUSELED_MASK) != 0) { (void)pel_error("IN USE LED not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; USER_MASK) != USER_MASK) { (void)pel_error("USER TRIGGER bit not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; KEEPALIVE_MASK) != KEEPALIVE_MASK) { (void)pel_error("KEEPALIVE bit not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; CLK10MHZ_MASK) != CLK10MHZ_MASK) { (void)pel_error("CLK 10MHZ bit not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; CLK100KHZ_MASK) != CLK100KHZ_MASK) { (void)pel_error("CLK 100KHZ bit not functional.\n"); errors++; } } /*  * 3. Verify USER=0, INUSE LED=1 and KEEPALIVE=0  */ pel-&gt;car.bit.is_new_led = 0; if (pel_play_pattern_string("x222z227", 0) != SUCCESS) { (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return (Failure); } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; INUSELED_MASK) != INUSELED_MASK) { (void)pel_error("IN USE LED not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; USER_MASK) != 0) { (void)pel_error("USER TRIGGER bit not functional.\n"); errors++; } if ((pel-&gt;magic_chip[0].m.logic_sample &amp; KEEPALIVE_MASK) != 0) { (void)pel_error("KEEPALIVE bit not functional.\n"); errors++; } } /*  * 4. Verify that CLK100KHZ toggles  */ attempts = 0; found_a_0 = FALSE; found_a_1 = FALSE; while (++attempts &lt;= MAX_CLK100KHZ_ATTEMPTS &amp;&amp; (!found_a_0    !found_a_1)) { if (diag_play() != SUCCESS) { (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return (Failure); } if ((pel-&gt;magic_chip[0].m.ttl_sample &amp; CLK100KHZ_MASK) == 0) { found_a_0 = TRUE; } else { found_a_1 = TRUE; } } if (--attempts == MAX_CLK100KHZ_ATTEMPTS) { (void)lm_error("CLK 100KHZ bit not functional in %d attempts.\n", attempts); errors++; } else { (void)lm_message("It took %d attempts to verify that CLK 100KHZ toggles.\n", attempts); } } /*  * 5. Verify that CLK10MHZ is approximately the right frequency  */ /* first see if CLK 10MHZ toggles */ attempts = 0; found_a_0 = FALSE; found_a_1 = FALSE; while (++attempts &lt;= MAX_CLK10MHZ_ATTEMPTS &amp;&amp; (!found_a_0    !found_a_1)) { if (diag_play() != SUCCESS) { (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return (Failure); } if ((pel-&gt;magic_chip[0].m.ttl_sample &amp; CLK10MHZ_MASK) == 0) { found_a_0 = TRUE; } else { found_a_1 = TRUE; } } if (--attempts == MAX_CLK10MHZ_ATTEMPTS) { (void)lm_error("CLK 10MHZ bit not functional in %d attempts.\n", attempts); errors++; } else { (void)lm_message("It took %d attempts to verify that CLK 10MHZ toggles.\n", attempts); } } /*  * Measure CLK 10MHZ frequency crudely  *  * Note: It is important that the keepalive clocks get turned off before they get turned on  * to verify that they start up both quickly and correctly.  */ attempts = 0; differences = 0; for (attempts = 0; attempts &lt; MAX_CLK10MHZ_ATTEMPTS; attempts++) { pel-&gt;car.bit.private = 1; if (pel_play_pattern_string("x222z227", 0) != SUCCESS) { /* turn off clocks */ (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return (Failure); } lm_delay(1); /* make keepalive clocks stay off for a while */ pel-&gt;car.bit.private = 0; if (pel_play_pattern_string("x222z227", 0) != SUCCESS) { /* turn on clocks */ (void)pel_error("Unable to play keepalive/trigger bit test pattern.\n"); return (Failure); } if ((pel-&gt;magic_chip[0].m.ttl_sample &amp; CLK10MHZ_MASK) != (pel-&gt;magic_chip[0].m.logic_sample &amp; CLK10MHZ_MASK)) { differences++; } } </pre>			



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_crc.c	DATE 5/23/89	PAGE # 9/118
TIME 4:41:26 pm				
LINE #	SOURCE TEXT			
961	}			
962	}			
963	if ((differences = 100) / attempts > 50) {			
964	pel_error("CLK LHM2 is off frequency\n");			
965	errors++;			
966	}			
967	lm_message("For CLK LHM2 found %d differences in %d attempts\n", differences, attempts);			
968				
969	return errors > FAILURE ? FAILURE : SUCCESS;			
970	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_edge.c	DATE 5/23/89	PAGE # 1/119
LINE #		SOURCE TEXT		
1	2	/* SOC5 ID: pel_edge.c rev 3.1, 4/24/89 at 07:49:55 */		
3	4	/*.....		
4	5	/*.....		
5	6	/*.....		
6	7	/*.....		
7	8	/*.....		
8	9	/*.....		
9	10	/*.....		
10	11	/*.....		
11	12	/*.....		
12	13	/*.....		
13	14	/*.....		
14	15	/*.....		
15	16	/*.....		
16	17	/*.....		
17	18	/*.....		
18	19	/*.....		
19	20	/*.....		
20	21	/*.....		
21	22	/*.....		
22	23	/*.....		
23	24	/*.....		
24	25	/*.....		
25	26	/*.....		
26	27	/*.....		
27	28	/*.....		
28	29	/*.....		
29	30	/*.....		
30	31	/*.....		
31	32	/*.....		
32	33	/*.....		
33	34	/*.....		
34	35	/*.....		
35	36	/*.....		
36	37	/*.....		
37	38	/*.....		
38	39	/*.....		
39	40	/*.....		
40	41	/*.....		
41	42	/*.....		
42	43	/*.....		
43	44	/*.....		
44	45	/*.....		
45	46	/*.....		
46	47	/*.....		
47	48	/*.....		
48	49	/*.....		
49	50	/*.....		
50	51	/*.....		
51	52	/*.....		
52	53	/*.....		
53	54	/*.....		
54	55	/*.....		
55	56	/*.....		
56	57	/*.....		
57	58	/*.....		
58	59	/*.....		
59	60	/*.....		
60	61	/*.....		
61	62	/*.....		
62	63	/*.....		
63	64	/*.....		
64	65	/*.....		
65	66	/*.....		
66	67	/*.....		
67	68	/*.....		
68	69	/*.....		
69	70	/*.....		
70	71	/*.....		
71	72	/*.....		
72	73	/*.....		
73	74	/*.....		
74	75	/*.....		
75	76	/*.....		
76	77	/*.....		
77	78	/*.....		
78	79	/*.....		
79	80	/*.....		
80	81	/*.....		
81	82	/*.....		
82	83	/*.....		
83	84	/*.....		
84	85	/*.....		
85	86	/*.....		
86	87	/*.....		
87	88	/*.....		
88	89	/*.....		
89	90	/*.....		
90	91	/*.....		
91	92	/*.....		
92	93	/*.....		
93	94	/*.....		
94	95	/*.....		
95	96	/*.....		
96	97	/*.....		
97	98	/*.....		
98	99	/*.....		
99	100	/*.....		
100	101	/*.....		
101	102	/*.....		
102	103	/*.....		
103	104	/*.....		
104	105	/*.....		
105	106	/*.....		
106	107	/*.....		
107	108	/*.....		
108	109	/*.....		
109	110	/*.....		
110	111	/*.....		
111	112	/*.....		
112	113	/*.....		
113	114	/*.....		
114	115	/*.....		
115	116	/*.....		
116	117	/*.....		
117	118	/*.....		
118	119	/*.....		
119	120	/*.....		

[illegible]

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_edge.c	DATE 5/23/89	PAGE # 3/121
LINE #		SOURCE TEXT		
241		return_status = FAILURE;		
242		if (lm_error(format,		
243		current_lane + 'A', current_pel,		
244		0, 1, soft_driver_test->driver) != SUCCESS) {		
245		return(FAILURE);		
246		}		
247		for (magic = 1; magic < 5; magic++) {		
248		pattern_word_rotate(4);		
249		pattern_word_rotate(5);		
250		pattern_word_rotate(6);		
251		pattern_word_rotate(7);		
252		pattern_word_rotate(8);		
253		pattern_word_rotate(9);		
254		pattern_word_rotate(10);		
255		pattern_word_rotate(11);		
256		pattern_word_rotate(12);		
257		pattern_word_rotate(13);		
258		if (diag_play() == FAILURE) {		
259		(void)pel_error("Unable to play driver test pattern.\n");		
260		return(FAILURE);		
261		}		
262		if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */		
263		return_status = FAILURE;		
264		if (lm_error(format,		
265		current_lane + 'A', current_pel,		
266		magic, (magic + 1) % 5,		
267		soft_driver_test->driver) != SUCCESS) {		
268		return(FAILURE);		
269		}		
270		}		
271		}		
272		lm_message("done.\n");		
273		/*		
274		* Test all medium drivers		
275		*/		
276		if (tmg_set_timing(period, edgetime, 01, 1000001, 10000001) != SUCCESS) {		
277		return FAILURE;		
278		}		
279		lm_message("Testing Medium Drivers");		
280		for (medium_driver_test = Medium_Driver_Tests, medium_driver_test->driver, ++medium_driver_test) {		
281		lm_message("-");		
282		pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = medium_driver_test->result;		
283		if (pel_play_pattern_string(medium_driver_test->pattern, 0) != SUCCESS) {		
284		(void)pel_error("Unable to play driver test pattern.\n");		
285		return (FAILURE);		
286		}		
287		if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */		
288		return_status = FAILURE;		
289		if (lm_error("Lane 4c pel td: Magic chip 0 failed ts test.\n",		
290		current_lane + 'A', current_pel,		
291		medium_driver_test->driver) != SUCCESS) {		
292		return(FAILURE);		
293		}		
294		for (magic = 1; magic < 5; magic++) {		
295		pattern_word_rotate(4);		
296		pattern_word_rotate(5);		
297		pattern_word_rotate(6);		
298		pattern_word_rotate(7);		
299		pattern_word_rotate(8);		
300		pattern_word_rotate(9);		
301		pattern_word_rotate(10);		
302		pattern_word_rotate(11);		
303		pattern_word_rotate(12);		
304		pattern_word_rotate(13);		
305		pattern_word_rotate(14);		
306		pattern_word_rotate(15);		
307		pattern_word_rotate(16);		
308		pattern_word_rotate(17);		
309		pattern_word_rotate(18);		
310		pattern_word_rotate(19);		
311		pattern_word_rotate(20);		
312		pattern_word_rotate(21);		
313		pattern_word_rotate(22);		
314		pattern_word_rotate(23);		
315		if (diag_play() == FAILURE) {		
316		(void)pel_error("Unable to play driver test pattern.\n");		
317		return(FAILURE);		
318		}		
319		if (compare_magic_chips(11, pattern) != SUCCESS) { /* value */		
320		return_status = FAILURE;		
321		if (lm_error("Lane 4c pel td: Magic chip 0 failed ts test.\n",		
322		current_lane + 'A', current_pel,		
323		magic,		
324		medium_driver_test->driver) != SUCCESS) {		
325		return(FAILURE);		
326		}		
327		}		
328		lm_message("done.\n");		
329		return(return_status);		
330		}		
331		}		

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_err.c

DATE

5/23/89

PAGE #

TIME 4:41:27 pm

1/122

```

1  /* SOCS_ID: pel_err.c rev 3.1, 4/24/89 at 07:49:38 */
2
3  #include "common.h"
4  #include "mod_def.h"
5  #include "modular_extn.h"
6  #include "magic.h"
7  #include "pel.h"
8
9  pel_error_test()
10 {
11     /* This test simulates the insertion and removal of an adapter. */
12     /* On the diagnostic dab, KEIN is connected to PRESENT. */
13     /* If KEIN is high, adapter is removed. */
14     /* If KEIN is low, adapter is inserted. */
15     /* Check 1 (reset):      car = 0x22 */
16     /* Check 2 (insertion):  car = 0x02 */
17     /* Check 3 (initialization): car = 0x12 */
18     /* Check 4 (removal):    car = 0x13 */
19     /* Check 5 (play error): car = 0x22 */
20
21     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
22     register long retval = SUCCESS;
23     char *error_format =
24         "%s:playerror--td error=%td present=%td active=%td\n";
25
26     if (diag_clear_errors() != SUCCESS)
27         return FAILURE;
28
29     if (pel_lane_init() != SUCCESS) { /* set up pac/tmg/pel/dab */
30         (void) lm_error("Unable to initialize lane %c",
31             current_lane + 'A');
32         return FAILURE;
33     }
34
35     pel->car.reg = 0; /* start in a known (default) state */
36     pel->car.bit.espram_in = 1; /* diag dab not present */
37     pel_clear_pel_errors();
38     lm_message("Check 1 (reset):      car = 0x%x", pel->car.reg);
39     if ((pel->car.reg) != 0x22) {
40         (void) lm_error(error_format, "Reset check",
41             pel->car.bit.play_error, 1,
42             pel->car.bit.error, 1,
43             pel->car.bit.present, 0,
44             pel->car.bit.active, 0);
45         retval = FAILURE;
46     }
47     pel->car.bit.espram_in = 0; /* diag dab present */
48     lm_message("Check 2 (insertion):  car = 0x%x", pel->car.reg);
49     if ((pel->car.reg) != 0x02) {
50         (void) lm_error(error_format, "Insertion check",
51             pel->car.bit.play_error, 1,
52             pel->car.bit.error, 0,
53             pel->car.bit.present, 1,
54             pel->car.bit.active, 0);
55         retval = FAILURE;
56     }
57     pel->car.bit.reset = 0; /* reset */
58     pel->car.bit.initialise = 0;
59     pel->car.bit.initialise = 1;
60     pel->car.bit.reset = 1; /* lift reset */
61     lm_message("Check 3 (initialization): car = 0x%x", pel->car.reg);
62     if ((pel->car.reg) != 0x12) {
63         (void) lm_error(error_format, "Initialization check",
64             pel->car.bit.play_error, 1,
65             pel->car.bit.error, 1,
66             pel->car.bit.present, 1,
67             pel->car.bit.active, 1);
68         retval = FAILURE;
69     }
70     pel->car.bit.espram_in = 1; /* diag dab not present */
71     lm_message("Check 4 (removal):    car = 0x%x", pel->car.reg);
72     if ((pel->car.reg) != 0x13) {
73         (void) lm_error(error_format, "Removal check",
74             pel->car.bit.play_error, 1,
75             pel->car.bit.error, 0,
76             pel->car.bit.present, 0,
77             pel->car.bit.active, 0);
78         retval = FAILURE;
79     }
80
81     pel->car.bit.initialise = 0;
82     pel->car.bit.reset = 0; /* reset */
83     pel->car.bit.reset = 1; /* lift reset */
84     if (pel_play_pattern_string("111111", 0) != SUCCESS) {
85         lm_message("ERROR playing pattern string");
86         return FAILURE;
87     }
88     lm_message("Check 5 (play error):  car = 0x%x", pel->car.reg);
89     if ((pel->car.reg) != 0x22) {
90         (void) lm_error(error_format, "Play error check",
91             pel->car.bit.play_error, 0,
92             pel->car.bit.error, 0,
93             pel->car.bit.present, 0,
94             pel->car.bit.active, 0);
95         retval = FAILURE;
96     }
97
98     pel_clear_pel_errors();
99     return retval;
100 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_id.c	DATE 5/23/89 TIME 4:41:27 pm	PAGE # 1/123
LINE #	SOURCE TEXT			
1	/* SCOS_ID: pel_id.c rev 3.1, 4/24/89 at 07:50:08 */			
2	#include "common.h"			
3	#include "in_diags.h"			
4	#include "id.h"			
5	#include "magic.h"			
6	#include "pel.h"			
7				
8	/*			
9	* Compute and verify checksum on ID PROM			
10	* Pass pointer to first byte in ID PROM.			
11	* Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart.			
12	* Algorithm:			
13	*     Initialize checksum to an arbitrary (but well-known) value			
14	*     for each byte (including checksum)			
15	*         circular left shift left checksum			
16	*         add data byte			
17	*         mask checksum to 8 bits			
18	* Returns computed checksum.			
19	*/			
20	int			
21	pel_id_check(pel)			
22	{			
23	register int checksum;			
24	register unsigned byte_count;			
25	checksum= ID_CHECKSUM_INIT;			
26	for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)			
27	{			
28	checksum= (checksum << 1) + ((checksum & 0x80) >> 7);			
29	checksum= (int)pel->idprom.id_prom[byte_count].data;			
30	checksum= 0xFF;			
31	}			
32	return(checksum);			
33	}			
34	/*			
35	* Load an ID PROM into a character buffer			
36	* address is the physical byte address of the first byte in the ID PROM.			
37	* buffer is the byte address of the first byte in a ID_NUM_BYTES buffer.			
38	* In practice, buffer is really an appropriate ID_PROM_XXX structure, and			
39	* a pointer to it is passed, cast to (u_char *).			
40	*/			
41	void			
42	pel_id_load(pel, buffer)			
43	{			
44	register int byte_count;			
45	for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)			
46	{			
47	*buffer= (unsigned char)pel->idprom.id_prom[byte_count].data;			
48	buffer++;			
49	}			
50	}			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_magic.c

DATE

5/23/89

PAGE #

1/124

TIME

4:41:27 pm

```

LINE # SOURCE TEXT
1  /* SCCS ID: pel_magic.c rev 3.1.1, 4/24/89 at 07:50:11 */
2
3  .....
4  .....
5  .....
6  .....
7  .....
8  .....
9  .....
10 .....
11 #include "math.h"
12 #include "common.h"
13 #include "mod_def.h"
14 #include "moduler_extn.h"
15 #include "vrtx.h"
16 #include "lm_diags.h"
17 #include "tmg.h"
18 #include "tmg_extn.h"
19 #include "pac.h"
20 #include "pac_def.h"
21 #include "pac_extn.h"
22 #include "magic.h"
23 #include "pel.h"
24 #include "pel_preamble.h"
25
26 #define print_magic_register(reg)
27 int reg;
28 {
29     register PEL *pel
30     = (PEL *) (pel_addr(current_lane, current_pel));
31     register long chip;
32
33     for (chip = 4; chip >= 0; chip--) {
34         lm_message("04x ", pel->magic_chip[chip].reg(reg));
35     }
36     lm_message("\n");
37 }
38
39 #define print_magic_state()
40 {
41     lm_message("DATA OUT (0) = ", print_magic_register(0));
42     lm_message("RDOC (1) = ", print_magic_register(1));
43     lm_message("RDOC (2) = ", print_magic_register(2));
44     lm_message("CYCOUT (3) = ", print_magic_register(3));
45     lm_message("LCYC (4) = ", print_magic_register(4));
46     lm_message("RDOEN (5) = ", print_magic_register(5));
47     lm_message("DATA IN (6) = ", print_magic_register(6));
48     lm_message("ERROR DATA (7) = ", print_magic_register(7));
49     lm_message("SPORT (8) = ", print_magic_register(8));
50     lm_message("U (9) = ", print_magic_register(9));
51     lm_message("MIZ (10) = ", print_magic_register(10));
52     lm_message("VALUE (11) = ", print_magic_register(11));
53     lm_message("TIMING (12) = ", print_magic_register(12));
54     lm_message("STATUS (14) = ", print_magic_register(14));
55 }
56
57 #define magic_circular_shift()
58 {
59     register PEL *pel
60     = (PEL *) (pel_addr(current_lane, current_pel));
61     register u_short *car = &(pel->car.reg);
62     PAT_WORD patterns;
63
64     *car = 0; /* MUST keep the PEL reset during this test */
65
66     if (!pel_load_pattern_string("z1z1z1z5", 0, STOP_MODE)) {
67         pel_error("Could not load pattern string\n");
68         return (FAILURE);
69     }
70
71     pel_read_pattern(3, &patterns);
72     patterns.pattern.data[0] = pel->magic_chip[4].reg[3];
73     patterns.pattern.data[1] = pel->magic_chip[3].reg[3];
74     patterns.pattern.data[2] = pel->magic_chip[2].reg[3];
75     patterns.pattern.data[3] = pel->magic_chip[1].reg[3];
76     patterns.pattern.data[4] = pel->magic_chip[0].reg[3];
77     pel_write_pattern(3, &patterns);
78
79     if (diag_play() == FAILURE) {
80         pel_error("Could not play pattern\n");
81         pac_play_cleanup();
82         return (FAILURE);
83     }
84     return (SUCCESS);
85 }
86
87 #define magic_swap()
88 {
89     register PEL *pel
90     = (PEL *) (pel_addr(current_lane, current_pel));
91     register u_short *car = &(pel->car.reg);
92
93     *car = 0; /* MUST keep the PEL reset during this test */
94
95     if (!pel_load_pattern_string("z1z1z1z1", 0, STOP_MODE)) {
96         pel_error("Could not load pattern string\n");
97         return (FAILURE);
98     }
99
100     if (diag_play() == FAILURE) {
101         pel_error("Could not play pattern\n");
102         pac_play_cleanup();
103         return (FAILURE);
104     }
105     return (SUCCESS);
106 }
107
108 char wmc_s1[] = "z1z5555d5ASMS5CSDSE5z5a5u5f1z5a5u50515253545f5555d5z1";
109 char wmc_s2[] = "z1z1z1z5";
110 char wmc_s3[] = "z1z1z1z1";
111
112 struct wr_magic_ctl {
113     char *name;
114     char *string;
115     u_short patterns[5];
116 } wr_magic_ctl_data[] = {
117     "C1OUT", wmc_s1, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
118     "RESET<0>", wmc_s2, 0x5555, 0x5555, 0x5555, 0x5555, 0x5555,
119     "RESET<1>", wmc_s2, 0xcdef, 0x3210, 0x7654, 0xba98, 0xfedc,
120     "SET<0>", wmc_s2, 0x0000, 0xffff, 0xffff, 0xffff, 0xffff,
121     "SET<1>", wmc_s2, 0xffff, 0x0000, 0xffff, 0xffff, 0xffff,

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_magic.c		DATE 5/23/89	PAGE # 2/125
LINE #		SOURCE TEXT			
121		"DOFF" wmc_32, 0xffff, 0xffff, 0x0000, 0xffff, 0xffff,			
122		"DOIS" wmc_32, 0xffff, 0xffff, 0xffff, 0x0000, 0xffff,			
123		"ICTOW" wmc_32, 0xffff, 0xffff, 0xffff, 0xffff, 0x0000,			
124		"ICTOW" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
125		"SEOD" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
126		"SEOD" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
127		"FORMAT(0)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
128		"FORMAT(1)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
129		"TOGGLE" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
130		"SDLEN(0)" wmc_32, 0xffff, 0x0000, 0x0000, 0x0000, 0x0000,			
131		"SDLEN(1)" wmc_32, 0x0000, 0xffff, 0x0000, 0x0000, 0x0000,			
132		"SDLEN(2)" wmc_32, 0x0000, 0x0000, 0xffff, 0x0000, 0x0000,			
133		"SDLEN(3)" wmc_32, 0x0000, 0x0000, 0x0000, 0xffff, 0x0000,			
134		"SDLEN(0)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0xffff,			
135		"SDLEN(1)" wmc_32, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,			
136		"SDLEN(2)" wmc_32, 0x5555, 0x5555, 0x5555, 0x5555, 0x5555,			
137		"SDLEN(3)" wmc_32, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,			
138		0			
139		}			
140		}			
141		#ifdef info			
142		string format dodededodod... where d = data, c = control			
143		the shift register is x dump by 80 wide (5 chips X 16 bits).			
144		#endif info			
145		{			
146		pel_wr_magic_ctl()			
147		{			
148		register PEL *pel			
149		= (PEL *) (pel_addr(current_lane, current_pel));			
150		register u_short *car = &(pel->car_reg);			
151		register struct wr_magic_ctl *wmcop;			
152		int errors = 0;			
153		{			
154		if (diag_clear_errors() != SUCCESS)			
155		return (FAILURE);			
156		{			
157		if (pel_lane_init() != SUCCESS) {			
158		(void) pel_error("Pia Electronics lane init fails.\n");			
159		return FAILURE;			
160		}			
161		{			
162		*car = 0; /* MUST keep the PEL reset during this test */			
163		{			
164		for (wmcop = wr_magic_ctl_data, wmcop->name; ++wmcop; ) {			
165		if (pel_play_pattern_string(wmcop->string, 0) != SUCCESS) {			
166		in_error("Lane to Pel to pattern play failed\n",			
167		current_lane + 'A', current_pel,			
168		wmcop->name);			
169		return (FAILURE);			
170		}			
171		if (compare_magic_chips(3, wmcop->pattern) != SUCCESS) {			
172		in_error("Lane to Pel to compare failed\n",			
173		current_lane + 'A', current_pel,			
174		wmcop->name);			
175		++errors;			
176		}			
177		{			
178		if (errors > 0)			
179		return (FAILURE);			
180		else			
181		return (SUCCESS);			
182		}			
183		{			
184		display_magic_control(),			
185		{			
186		register struct wr_magic_ctl *wmcop;			
187		register int count;			
188		{			
189		setup_good_sample(tmg_measure_period());			
190		{			
191		wmcop = wr_magic_ctl_data;			
192		for (count = 0; count < 11; ++wmcop, ++count) {			
193		in_message("t-10- ", wmcop->name);			
194		print_magic_register(3);			
195		magic_circular_shift();			
196		}			
197		{			
198		magic_swap();			
199		for (count = 0; count < 11; ++wmcop, ++count) {			
200		in_message("t-10- ", wmcop->name);			
201		print_magic_register(3);			
202		magic_circular_shift();			
203		}			
204		magic_swap();			
205		return (SUCCESS);			
206		}			
207		{			
208		/* compare_magic_chips(reg, pattern) */			
209		/* pattern is a 30 character string representing the 80 bits of expected data */			
210		/* "44443333222211110000" */			
211		{			
212		compare_magic_chips(reg, pattern)			
213		u_short *pattern;			
214		{			
215		register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));			
216		char *format = "magic chip %d, Reg %d got %04x, expected %04x\n";			
217		register u_short result;			
218		register long chip_errors;			
219		u_short errbits;			
220		int i;			
221		char buffer[256], buf[8];			
222		{			
223		errors = 0;			
224		{			
225		for (chip = 0; chip < 5; ++chip) {			
226		if ((result = pel->magic_chip[chip].reg[reg]) != pattern[chip]) {			
227		(void) in_error(format, chip, reg, result, pattern[chip]);			
228		errors++;			
229		{			
230		errbits = result ^ pattern[chip];			
231		strcpy(buffer, "Check pattern pin(s):\n");			
232		for (i = 0; i < 16; ++i) {			
233		if (errbits & (1 << i)) {			
234		sprintf(buf, "%d", chip * 16 + i);			
235		strcat(buffer, buf);			
236		}			
237		{			
238		in_error("%s\n", buffer);			
239		{			
240		{			



1105

5,353,243

1106

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/peL_magic.c	DATE 5/23/89	PAGE # 3/126
TIME 4:41:27 pm				
LINE #	SOURCE TEXT			
241	)			
242				
243	if (errors != 0) {			
244	return(FAILURE);			
245	} else {			
246	return(SUCCESS);			
247	}			
248	)			



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_opcode.c

DATE 5/23/89 PAGE #  
TIME 4:41:28 pm 2/128

```

121  if (compare_magic_chips(0, pattern) != SUCCESS) {
122      (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
123      (DATA_OUT)\n");
124      errors++;
125      pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
126      if (compare_magic_chips(3, pattern) != SUCCESS) {
127          (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
128          (CTL_OUT)\n");
129          errors++;
130      }
131      pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
132      if (compare_magic_chips(5, pattern) != SUCCESS) {
133          (void)pel_error("    PEL (enabled) opcodes 000 and 001 failure \
134          (ENDEM)\n");
135          errors++;
136      }
137  }
138
139  /*
140  * 4. Test opcodes 010 with PEL enabled and but not selected for current operation.
141  */
142  lm_message("Testing PEL opcode 010 with PEL enabled and but not selected.\n");
143  if (pel_play_pattern_string("f2f2z7fc", 0) != SUCCESS) {
144      (void)pel_error("Failure to play OPCODE test pattern");
145      return (FAILURE);
146  }
147  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x0000; /* check DATA_OUT */
148  if (compare_magic_chips(0, pattern) != SUCCESS) {
149      (void)pel_error("    PEL (enabled) opcode 010 failure (DATA_OUT)\n");
150      errors++;
151  }
152  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
153  if (compare_magic_chips(3, pattern) != SUCCESS) {
154      (void)pel_error("    PEL (enabled) opcode 010 failure (CTL_OUT)\n");
155      errors++;
156  }
157  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
158  if (compare_magic_chips(5, pattern) != SUCCESS) {
159      (void)pel_error("    PEL (enabled) opcode 010 failure (ENDEM)\n");
160      errors++;
161  }
162  }
163
164  /*
165  * 5. Test opcodes 011 and 100 with PEL enabled and but not selected for current operation.
166  */
167  lm_message("Testing PEL opcodes 011 and 100 with PEL enabled and but not selected.\n");
168  if (pel_play_pattern_string("f2f2z7fc", 0) != SUCCESS) {
169      (void)pel_error("Failure to play OPCODE test pattern");
170      return (FAILURE);
171  }
172  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xffff; /* check DATA_OUT */
173  if (compare_magic_chips(0, pattern) != SUCCESS) {
174      (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
175      (DATA_OUT)\n");
176      errors++;
177  }
178  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
179  if (compare_magic_chips(3, pattern) != SUCCESS) {
180      (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
181      (CTL_OUT)\n");
182      errors++;
183  }
184  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
185  if (compare_magic_chips(5, pattern) != SUCCESS) {
186      (void)pel_error("    PEL (enabled) opcodes 011 and 100 failure \
187      (ENDEM)\n");
188      errors++;
189  }
190  }
191
192  /*
193  * 6. Test opcodes 101 with PEL enabled and but not selected for current operation.
194  */
195  lm_message("Testing PEL opcode 101 with PEL enabled and but not selected.\n");
196  if (pel_play_pattern_string("f2f2z7fc", 0) != SUCCESS) {
197      (void)pel_error("Failure to play OPCODE test pattern");
198      return (FAILURE);
199  }
200  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x0000; /* check DATA_OUT */
201  if (compare_magic_chips(0, pattern) != SUCCESS) {
202      (void)pel_error("    PEL (enabled) opcode 101 failure (DATA_OUT)\n");
203      errors++;
204  }
205  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
206  if (compare_magic_chips(3, pattern) != SUCCESS) {
207      (void)pel_error("    PEL (enabled) opcode 101 failure (CTL_OUT)\n");
208      errors++;
209  }
210  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
211  if (compare_magic_chips(5, pattern) != SUCCESS) {
212      (void)pel_error("    PEL (enabled) opcode 101 failure (ENDEM)\n");
213      errors++;
214  }
215  }
216
217  /*
218  * 7. Test opcodes 110 and 111 with PEL enabled and but not selected for current operation.
219  */
220  lm_message("Testing PEL opcodes 110 and 111 with PEL enabled and but not selected.\n");
221  if (pel_play_pattern_string("f2f2z7fc", 0) != SUCCESS) {
222      (void)pel_error("Failure to play OPCODE test pattern");
223      return (FAILURE);
224  }
225  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xffff; /* check DATA_OUT */
226  if (compare_magic_chips(0, pattern) != SUCCESS) {
227      (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \
228      (DATA_OUT)\n");
229      errors++;
230  }
231  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0xaaaa; /* check CTL_OUT */
232  if (compare_magic_chips(3, pattern) != SUCCESS) {
233      (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \
234      (CTL_OUT)\n");
235      errors++;
236  }
237  pattern[4] = pattern[3] = pattern[2] = pattern[1] = pattern[0] = 0x5555; /* check ENDEM */
238  if (compare_magic_chips(5, pattern) != SUCCESS) {
239      (void)pel_error("    PEL (enabled) opcodes 110 and 111 failure \
240

```

1111

5,353,243

1112

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_opcode.c	DATE 5/23/89	PAGE # 3/129
TIME 4:41:28 pm				
LINE #	SOURCE TEXT			
241	{BNDEN*}\n");			
242	errors++;			
243	}			
244				
245	return errors ? FAILURE : SUCCESS;			
246	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_pattern.c	DATE 5/23/89	PAGE # 1/130
LINE #	SOURCE TEXT			
1	/* SCCS ID: pel_pattern.c Rev 3.1, 4/24/89 at 07:58:13 */			
2	/*			
3	* Pattern play routines			
4	* used in PEL diagnosis			
5	*/			
6	.....			
7	.....			
8	.....			
9	.....			
10	#include <math.h>			
11	#include "common.h"			
12	#include "mod_def.h"			
13	#include "modeler_extn.h"			
14	#include "vtx.h"			
15	#include "lm_diags.h"			
16	#include "tag.h"			
17	#include "tag_extn.h"			
18	#include "pac.h"			
19	#include "pac_def.h"			
20	#include "pac_extn.h"			
21	#include "magic.h"			
22	#include "pel.h"			
23	#include "pel_preamble.h"			
24	.....			
25	.....			
26	.....			
27	.....			
28	.....			
29	.....			
30	.....			
31	.....			
32	.....			
33	.....			
34	.....			
35	.....			
36	.....			
37	.....			
38	.....			
39	.....			
40	.....			
41	.....			
42	.....			
43	.....			
44	.....			
45	.....			
46	.....			
47	.....			
48	.....			
49	.....			
50	.....			
51	.....			
52	.....			
53	.....			
54	.....			
55	.....			
56	.....			
57	.....			
58	.....			
59	.....			
60	.....			
61	.....			
62	.....			
63	.....			
64	.....			
65	.....			
66	.....			
67	.....			
68	.....			
69	.....			
70	.....			
71	.....			
72	.....			
73	.....			
74	.....			
75	.....			
76	.....			
77	.....			
78	.....			
79	.....			
80	.....			
81	.....			
82	.....			
83	.....			
84	.....			
85	.....			
86	.....			
87	.....			
88	.....			
89	.....			
90	.....			
91	.....			
92	.....			
93	.....			
94	.....			
95	.....			
96	.....			
97	.....			
98	.....			
99	.....			
100	.....			
101	.....			
102	.....			
103	.....			
104	.....			
105	.....			
106	.....			
107	.....			
108	.....			
109	.....			
110	.....			
111	.....			
112	.....			
113	.....			
114	.....			
115	.....			
116	.....			
117	.....			
118	.....			
119	.....			
120	.....			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_pattern.c	DATE 5/13/89	PAGE # 2/131
LINE #		SOURCE TEXT		
121		"Display pattern memory preamble (short)",		
122		display_pattern_preamble,		
123		LM_DIAG_utility,		
124		LM_DIAG_main,		
125		0		
126		{		
127		}		
128		"4",		
129		"Display pattern memory preamble (long)",		
130		display_pattern_preamble_verbosely,		
131		LM_DIAG_utility,		
132		LM_DIAG_main,		
133		0		
134		{		
135		}		
136		"5",		
137		"Display pattern string code",		
138		display_pattern_string_code,		
139		LM_DIAG_utility,		
140		LM_DIAG_main,		
141		0		
142		{		
143		}		
144		"6",		
145		"Build Preamble",		
146		build_preamble,		
147		LM_DIAG_utility,		
148		LM_DIAG_main,		
149		0		
150		{		
151		}		
152		static LM_DIAG_MENU menu =		
153		{		
154		"PEL UTILITIES",		
155		sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),		
156		menu_list		
157		}		
158		menu.title = parent_menu->		
159		menu.title = parent_menu->		
160		menu.items[parent_menu->current_selection].menu_text,		
161		return lm_display_menu(menu);		
162		}		
163		}		
164		#define MIN_PATTERN_LENGTH 4		
165		#define ODD_ERROR -1		
166		pel_pattern_count(pattern_string)		
167		char *pattern_string;		
168		{		
169		register long len;		
170		static char odd_error[] =		
171		"Pattern string has odd number of chars:\n\t\"%s\\n",		
172		{		
173		if ((len = strlen(pattern_string)) % 2) {		
174		(void)lm_error(odd_error, pattern_string);		
175		return ODD_ERROR;		
176		}		
177		return len >> 1; /* pattern count */		
178		}		
179		}		
180		}		
181		build_pattern_data(pattern_string, first_pattern_no)		
182		char *pattern_string;		
183		{		
184		struct string_pattern *pel_decode_string_patterns(),		
185		static char set_error[] =		
186		"Pattern memory set error at line = %d, address = %08x\\n",		
187		{		
188		PAT_WORD pattern;		
189		register long i;		
190		register u_long a, pattern_count;		
191		struct string_pattern *string_pattern;		
192		{		
193		if ((pattern_count = pel_pattern_count(pattern_string)) == ODD_ERROR) {		
194		return (0);		
195		}		
196		}		
197		for (a = first_pattern_no; pattern_string != '\\0'; ++a) {		
198		/* set the pattern data according to first character in pair */		
199		if (!((string_pattern		
200		= pel_decode_string_pattern(pattern_string))) {		
201		lm_message("Bad pel_load_pattern_string\\n");		
202		return(0);		
203		}		
204		}		
205		for (i = 0; i < 5; i++)		
206		pattern.pattern.data[i] = string_pattern->data[i];		
207		pattern_string++; /* move to second pair of characters */		
208		/* compute pel_control and pel_address		
209		from second of character pair */		
210		if ((pattern_string > '\\0') && (pattern_string < '\\7')) {		
211		pattern.pattern.pel_control		
212		= pattern_string - '\\0';		
213		pattern.pattern.pel_address = current_pel;		
214		} else if ((pattern_string > '\\a') && (pattern_string < '\\h')) {		
215		pattern.pattern.pel_control		
216		= pattern_string - '\\a';		
217		pattern.pattern.pel_address		
218		= (current_pel + 1) & 16;		
219		} else {		
220		lm_message("Bad pel_load_pattern_string\\n");		
221		return(0);		
222		}		
223		/* Make sure the branch bits and the stop bit are zero (NOP) */		
224		pattern.pattern.branch = 0;		
225		pattern.pattern.stop = 0;		
226		/* Set the spare bits to zero so that pattern loading is */		
227		consistent. Clear the USER bit */		
228		pattern.pattern.backplane_spare = 0;		
229		pattern.pattern.pcc_spare = 0;		
230		pattern.pattern.user = 0;		
231		pattern_string++; /* move to next pair of characters */		
232		/* write the pattern to pattern memory,		
233		and move to next address */		
234		if (pel_write_pattern(a, pattern) != SUCCESS) {		
235		}		
236		}		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_pattern.c

DATE 5/23/89  
TIME 4:41:28 pm

PAGE #  
3/132

```

LINE # SOURCE TEXT
241      lm_message(Set_error, current_lane, n);
242      return(0);
243  }
244  }
245  return (patterns_count);
246  }
247
248  pel_type_pattern_string()
249  {
250      char patterns[256];
251      long start_patterns = 0, dab_type = 0;
252
253      patterns[0] = '\0';
254      lm_get_input("Enter pattern string: ", patterns, 256);
255      diag_get_long(&start_patterns, "start pattern number", 01,
256      (long)pac(current_lane).sum_patterns);
257      diag_get_long(&dab_type,
258      "Enter DAB type (Public_DAB = 0, PRIVATE_DAB = 1)", 01, 11);
259
260      if (set_vlog1(0.8) == FAILURE) return(FAILURE);
261      if (set_vlog2(2.0) == FAILURE) return(FAILURE);
262      if (set_vlog3(0.3) == FAILURE) return(FAILURE);
263      if (set_vlog4(0.4) == FAILURE) return(FAILURE);
264      if (set_vlog5(4.0) == FAILURE) return(FAILURE);
265      if (set_vlog6(4.4) == FAILURE) return(FAILURE);
266
267      if (pel_lane_init() != SUCCESS) { /* set up pac/seg/pel/dab */
268          (void)lm_error("Unable to initialize lane %c", current_lane + 'A');
269          return(FAILURE);
270      }
271      if (pel_dab_init(dab_type) != SUCCESS) {
272          (void)lm_error("Unable to initialize DAB %c, PEL id %c",
273          current_lane + 'A', current_pel);
274          return(FAILURE);
275      }
276  }
277
278  if (pel_play_pattern_string(patterns, start_patterns) == SUCCESS) {
279      print_magic_state();
280      return SUCCESS;
281  } else {
282      lm_message("ERROR playing pattern string\n");
283      return FAILURE;
284  }
285  }
286
287  pel_enter_pattern_string()
288  {
289      char patterns[256];
290      long start_patterns = 0;
291
292      patterns[0] = '\0';
293      lm_get_input("Enter pattern string: ", patterns, 256);
294      diag_get_long(&start_patterns, "start pattern number", 01,
295      (long)pac(current_lane).sum_patterns);
296
297      if (!pel_load_pattern_string(patterns, start_patterns, STOP_MODE)) {
298          (void)pel_error("pel_load_pattern_string failed\n");
299          return (FAILURE);
300      }
301      return (SUCCESS);
302  }
303
304  pel_play_pattern_string(pattern_string, first_pattern_no)
305  char *pattern_string;
306  {
307      if (!pel_load_pattern_string(pattern_string,
308      first_pattern_no, STOP_MODE)) {
309          (void)pel_error("pel_play_pattern_string failed\n");
310          return (FAILURE);
311      }
312      if (diag_play() == FAILURE) {
313          (void)pel_error("pel_play_pattern_string failed in diag_play\n");
314          return(FAILURE);
315      }
316      return (SUCCESS);
317  }
318
319  pel_load_pattern_string(pattern_string, first_pattern_no, mode)
320  char *pattern_string;
321  {
322      register u_long patterns_count;
323
324      if ((patterns_count = build_patterns_data(pattern_string,
325      first_pattern_no)) == 0) {
326          (void)pel_error("Can't load pattern data\n");
327          return 0;
328      }
329
330      if (pel_build_patterns_control(first_pattern_no, patterns_count, mode)
331      != SUCCESS) {
332          (void)pel_error("Can't load pattern control\n");
333          return(0);
334      }
335      return patterns_count;
336  }
337
338  struct string_patterns =
339  pel_decode_string_patterns(c)
340  char c;
341  {
342      struct string_patterns *p;
343
344      for (p = String_patterns; p->code; ++p) {
345          if (c == p->code) return p;
346      }
347      return 0; /* undefined code */
348  }
349
350  char
351  pel_encode_string_patterns(d)
352  short d[];
353  {
354      register int chip;
355      register struct string_patterns *p;
356
357      for (p = String_patterns; p->code; ++p) {
358          for (chip = 0; chip < 5; ++chip) {

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_pattern.c	DATE 5/23/89	PAGE # 4/133
LINE #		SOURCE TEXT	TIME 4:41:28 pm	
361		if (p->data[chip] != d[chip]) break;		
362		if (chip == 5) return p->code;		
363		return 0; /* undefined code */		
364		}		
365		display_patterns_preamble()		
366		{		
367		patterns_preamble(0);		
368		}		
369		display_patterns_preamble_verbose()		
370		{		
371		patterns_preamble(1);		
372		}		
373		set_patterns_preamble(start_patterns, preamble_array)		
374		{		
375		u_long start_patterns;		
376		preamble preamble_array[];		
377		{		
378		PAT_WORD patptr;		
379		u_long patterns, limit_patterns = start_patterns + PREAMBLE_SIZE;		
380		u_long pin, start_pin, limit_pin;		
381		u_long chip;		
382		for (pin = 0; pin < 80; ++pin) {		
383		preamble_array[pin].x = 0;		
384		}		
385		for (patterns = start_patterns; patterns < limit_patterns; ++patterns) {		
386		pel_read_patterns(patterns, &patptr);		
387		for (chip = 0; chip < 5; ++chip) {		
388		start_pin = (4 - chip) * 16;		
389		limit_pin = start_pin + 16;		
390		for (pin = start_pin; pin < limit_pin; ++pin) {		
391		if ((patptr.patterns.data[chip] & (1 << (pin & 16))) != 0)		
392		preamble_array[pin].x  =		
393		(1 << (patterns - start_patterns));		
394		}		
395		}		
396		return SUCCESS;		
397		}		
398		set_patterns_preamble(start_patterns, preamble_array)		
399		{		
400		u_long start_patterns;		
401		preamble preamble_array[];		
402		{		
403		PAT_WORD patptr;		
404		u_long patterns, limit_patterns;		
405		u_long pin, start_pin, limit_pin;		
406		u_long chip;		
407		limit_patterns = start_patterns + PREAMBLE_SIZE;		
408		for (patterns = start_patterns; patterns < limit_patterns; ++patterns) {		
409		pel_read_patterns(patterns, &patptr);		
410		for (chip = 0; chip < 5; ++chip) {		
411		patptr.patterns.data[chip] = 0;		
412		start_pin = (4 - chip) * 16;		
413		limit_pin = start_pin + 16;		
414		for (pin = start_pin; pin < limit_pin; ++pin) {		
415		if ((patptr.patterns.data[chip] & (1 << (pin & 16))) != 0)		
416		preamble_array[pin].x  =		
417		(1 << (patterns - start_patterns));		
418		}		
419		if ((preamble_array[pin].x & (1 << (patterns - start_patterns))) != 0)		
420		patptr.patterns.data[chip]  = (1 << (pin & 16));		
421		}		
422		pel_write_patterns(patterns, &patptr);		
423		}		
424		return SUCCESS;		
425		}		
426		patterns_preamble(verbose)		
427		{		
428		register int count;		
429		preamble preamble_array[80];		
430		u_long start_patterns;		
431		int pin;		
432		int chip;		
433		start_patterns = 0;		
434		diag_get_log(&start_patterns, "start patterns", start_patterns,		
435		(u_long)(pac(current_lane).num_patterns - 1));		
436		get_patterns_preamble(start_patterns, preamble_array);		
437		if (verbose) {		
438		for (count = pin = 0; pin < 80; ++pin, ++count) {		
439		if (count % 20) {		
440		if (!(count == more(pin, 80, count)))		
441		break;		
442		lm_message("pin %2d", pin);		
443		decode_preamble(preamble_array[pin].x);		
444		}		
445		} else {		
446		for (pin = 0; pin < 16; ++pin) {		
447		lm_message("pin %2d", pin);		
448		for (chip = 0; chip < 5; ++chip) {		
449		lm_message("pin %2d %07x",		
450		pin + 16 * chip, preamble_array[pin + 16 * chip].x);		
451		}		
452		lm_message("\n");		
453		}		
454		return SUCCESS;		
455		}		
456		static char *array_format[] =		
457		{		
458		"RC",		
459		"R2",		
460		"R1",		
461		"DNRL",		
462		};		
463		static char *array_toggle[] =		
464		{		
465		" ",		
466		"X",		
467		"0",		
468		"1",		
469		"2",		
470		"3",		
471		"4",		
472		"5",		
473		"6",		
474		"7",		
475		"8",		
476		"9",		
477		"0",		
478		"1",		
479		"2",		
480		"3",		



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_pattern.c	DATE 5/23/89	PAGE # 5/134
LINE #	SOURCE TEXT			
481	} "TOG",			
482	static char "array_doff[] =			
483	{			
484	"DOFF0",			
485	"DOFF5",			
486	},			
487	static char "array_hddis[] =			
488	{			
489	"HDDIS",			
490	},			
491	static char "array_lcycmd[] =			
492	{			
493	"LMD",			
494	},			
495	static char "array_lcychn[] =			
496	{			
497	"LMD",			
498	},			
499	static char "array_seqmd[] =			
500	{			
501	"SQMD",			
502	},			
503	static char "array_seqchn[] =			
504	{			
505	"SQMD",			
506	},			
507	static char "array_hmden[] =			
508	{			
509	"HMDEN",			
510	},			
511	static char "array_hmden[] =			
512	{			
513	"HMDEN",			
514	},			
515	},			
516	},			
517	},			
518	},			
519	decode_preamble(x)			
520	preamble x,			
521	{			
522	lm_message("to to SLEWAX KHEWAX Rtd Std to to to to to to to to\n",			
523	array_format(x.b.format),			
524	array_toggle(x.b.toggle),			
525	x.b.sclen,			
526	x.b.sclen,			
527	x.b.reset,			
528	x.b.set + 2,			
529	array_doff(x.b.doff),			
530	array_hddis(x.b.hddis),			
531	array_lcycmd(x.b.lcycmd),			
532	array_lcychn(x.b.lcychn),			
533	array_seqmd(x.b.seqmd),			
534	array_seqchn(x.b.seqchn),			
535	array_hmden(x.b.hmden),			
536	}			
537	build_preamble()			
538	{			
539	char reply(DIAG_MAX_INPUT), "s,			
540	preamble preamble_array[80],			
541	preamble x,			
542	u_long start_patterns,			
543	u_long pin, start_pin, end_pin,			
544	u_long format,			
545	u_long toggle,			
546	u_long sclen,			
547	u_long sclen,			
548	u_long reset,			
549	u_long set,			
550	u_long doff,			
551	u_long hddis,			
552	u_long lcycmd,			
553	u_long lcychn,			
554	u_long seqmd,			
555	u_long seqchn,			
556	u_long hmden,			
557	char "get_range()",			
558	start_pattern = 0,			
559	diag_get_ulong(&start_patterns, "start patterns", start_patterns,			
560	(u_long)(pcc(current_line).num_patterns - 1)),			
561	{			
562	lm_get_input("Enter pin list (P P.P. P-P.....) ",			
563	reply, (short)DIAG_MAX_INPUT),			
564	{			
565	if ((s = get_range(reply, &start_pin, &end_pin)) {			
566	lm_message("Although s = td and e = td\n", start_pin, end_pin),			
567	(void)pel_error("Illegal pin selection string\n"),			
568	return FAILURE;			
569	}			
570	if ((start_pin < 0)    (start_pin > 79)) {			
571	(void)lm_error("Illegal pin number td\n", start_pin),			
572	return FAILURE;			
573	}			
574	get_patterns_preamble(start_patterns, preamble_array);			
575	x.x = preamble_array[start_pin].x;			
576	format = x.b.format;			
577	toggle = x.b.toggle;			
578	sclen = (x.b.sclen) & 0xf;			
579	sclen = x.b.sclen;			
580	reset = x.b.reset;			
581	set = x.b.set + 2;			
582	doff = x.b.doff;			
583	hddis = (x.b.hddis) & 0xf;			
584	lcycmd = (x.b.lcycmd) & 0xf;			
585	lcychn = (x.b.lcychn) & 0xf;			
586	seqmd = (x.b.seqmd) & 0xf;			
587	seqchn = (x.b.seqchn) & 0xf;			
588	hmden = (x.b.hmden) & 0xf;			
589	diag_get_ulong(&format, "data format: 0->PC, 1->RZ, 2->R1, 3->DNRZ", 01, 31);			
590	diag_get_ulong(&toggle, "toggle", 01, 11);			
591	diag_get_ulong(&sclen, "sclen", 01, 151);			
592	diag_get_ulong(&reset, "reset edge", 01, 31);			
593	diag_get_ulong(&set, "set edge", 21, 51);			
594	diag_get_ulong(&doff, "driver off edge: 0->0, 1->5", 01, 11);			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_pattern.c	DATE 5/23/89	PAGE # 6/135
LINE #		SOURCE TEXT		
601		diag_get_ulong(&hddia, "hddia", 01, 11);		
602		diag_get_ulong(&lcycmd, "lcycmd", 01, 11);		
603		diag_get_ulong(&lcychd, "lcychd", 01, 11);		
604		diag_get_ulong(&seqmd, "seqmd", 01, 11);		
605		diag_get_ulong(&seqhd, "seqhd", 01, 11);		
606		diag_get_ulong(&hmden, "hmden", 01, 11);		
607		x.b.format = format;		
608		x.b.toggle = toggle;		
609		x.b.adlea = adlea;		
610		x.b.adhea = adhea;		
611		x.b.reset = reset;		
612		x.b.set = set - 2;		
613		x.b.doff = doff;		
614		x.b.hddia = hddia;		
615		x.b.lcycmd = lcycmd;		
616		x.b.lcychd = lcychd;		
617		x.b.seqmd = seqmd;		
618		x.b.seqhd = seqhd;		
619		x.b.hmden = hmden;		
620		lm_message("X\n", x.x);		
621		do {		
622		lm_message("d ... d\n", start_pin, end_pin);		
623		if ((start_pin < 0)    (start_pin > 79)) {		
624		(void)lm_error("Illegal pin number %d\n", start_pin);		
625		return FAILURE;		
626		}		
627		if ((end_pin < 0)    (end_pin > 79)) {		
628		(void)lm_error("Illegal pin number %d\n", end_pin);		
629		return FAILURE;		
630		}		
631		for (pin = start_pin; pin <= end_pin; ++pin) {		
632		preamble_array[pin].x = x.x;		
633		}		
634		} while (s = get_range(s, &start_pin, &end_pin));		
635		set_patterns_preamble(start_patterns, preamble_array);		
636		return SUCCESS;		
637		}		
638		char *preamble_bit_meaning[] = {		
639		"start preamble",		
640		"format",		
641		"toggle",		
642		"adlea",		
643		"adhea",		
644		"reset",		
645		"set",		
646		"doff",		
647		"hddia",		
648		"lcycmd",		
649		"lcychd",		
650		"seqmd",		
651		"seqhd",		
652		"hmden",		
653		"end preamble",		
654		};		
655		print_preamble_line_meaning(bit)		
656		{		
657		if ((bit >= 0) && (bit < PREAMBLE_SIZE)) {		
658		lm_message(preamble_bit_meaning[bit]);		
659		}		
660		display_patterns_string_codes()		
661		{		
662		char buffer[80];		
663		long start_patterns, end_patterns, this_end;		
664		int count, origia;		
665		start_patterns = 0;		
666		end_patterns = pac[current_lane].num_patterns - 1;		
667		diag_get_long(&start_patterns, "start patterns", start_patterns, end_patterns);		
668		diag_get_long(&end_patterns, "end patterns", start_patterns, end_patterns);		
669		count = 0;		
670		origia = start_patterns;		
671		for (this_end = start_patterns + 1; start_patterns < end_patterns;		
672		start_patterns += 32, this_end += 32) {		
673		if (this_end > end_patterns) this_end = end_patterns;		
674		lm_message("%10d: ", start_patterns);		
675		pel_get_patterns_string(start_patterns, this_end, buffer);		
676		lm_message("%s\n", buffer);		
677		if ((start_patterns < end_patterns) && !(++count % 20)) {		
678		if (!(count = more((int)(start_patterns - origia),		
679		(end_patterns - origia), count)))		
680		break;		
681		}		
682		return SUCCESS;		
683		}		
684		pel_get_patterns_string(start_patterns, end_patterns, s)		
685		u_long start_patterns, end_patterns;		
686		char *s;		
687		{		
688		FAT_WORD patptr;		
689		u_long patterns;		
690		for (pattern = start_patterns; pattern <= end_patterns; ++pattern) {		
691		pel_read_patterns(pattern, &patptr);		
692		if (!(*s = pel_encode_string_patterns(patptr.pattern.data)))		
693		*s = "?";		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_pattern.c

DATE  
5/23/89  
TIME  
4:41:28 pm

PAGE #  
7/136

LINE # SOURCE TEXT

```
721 ++;
722 *g++ = '0' + patptr->pattern.pel_control;
723 }
724 *s = '\0';
725 return SUCCESS;
726 }
727
728 display_pattern_memory()
729 {
730     register int i, count;
731     PAT_WORD patptr;
732     long start_pattern, stop_pattern, start_preamble;
733     char pattern_string[3];
734
735     start_pattern = 0;
736     stop_pattern = pec[current_lane].num_patterns - 1;
737     diag_get_long(&start_pattern, "start pattern", start_pattern, stop_pattern);
738     diag_get_long(&stop_pattern, "stop pattern", start_pattern, stop_pattern);
739
740     start_preamble = stop_pattern + 1; /* don't display preamble bits
741                                         if no preamble */
742     for (i = start_pattern, count = 0; i <= stop_pattern; ++i) {
743         pel_read_pattern(i, patptr)
744         in_message("10d: 1041 1042 1043 1044 1045 1046 ",
745                 i,
746                 patptr->pattern.data[0],
747                 patptr->pattern.data[1],
748                 patptr->pattern.data[2],
749                 patptr->pattern.data[3],
750                 patptr->pattern.data[4]);
751         in_message(" PAtt2d", patptr->pattern.pel_address);
752         in_message(" PCd", patptr->pattern.pel_control);
753         switch (patptr->pattern.branch) {
754             case 1: in_message(" BA"); break;
755             case 2: in_message(" BF"); break;
756             case 3: in_message(" BK"); break;
757             default: in_message(" "); break;
758         }
759         in_message("3s ", (patptr->pattern.stop)? "STOP": "");
760         in_message("3s ", (patptr->pattern.user)? "USER": "");
761         pel_get_pattern_string(i, i, pattern_string);
762         in_message("3s ", pattern_string);
763
764         if (patptr->pattern.pel_control == 2)
765             start_preamble = i; /* start of preamble */
766         print_preamble_line_meaning(i - start_preamble);
767         in_message("\n");
768         if ((i < stop_pattern) && (++count % 20) {
769             if (!(count == more(i - (1st)start_pattern,
770                                 stop_pattern - start_pattern, count)))
771                 break;
772         }
773     }
774 }
775
776 pel_read_pattern(pattern, patptr)
777 int pattern;
778 PAT_WORD *patptr;
779 {
780     extern u_long *Host_memory;
781     register u_long i, *mem;
782
783     if (!Host) return pec_read_pattern(pattern, patptr);
784
785     /* Check if pattern number is valid */
786     if((pattern < 0) || (pattern > 0x40000))
787         return(FAILURE);
788
789     mem = &(Host_memory[4 * pattern]);
790
791     /* Read pattern word */
792     for (i = 0; i < 3; ++i)
793         patptr->mem_bank[i] = *mem++;
794     return(SUCCESS);
795 }
796
797 pel_write_pattern(pattern, patptr)
798 int pattern;
799 PAT_WORD *patptr;
800 {
801     extern u_long *Host_memory;
802     register u_long i, *mem;
803
804     if (!Host) return pec_write_pattern(pattern, patptr);
805
806     /* Check if pattern number is valid */
807     if((pattern < 0) || (pattern > 0x40000))
808         return(FAILURE);
809
810     mem = &(Host_memory[4 * pattern]);
811
812     /* Read pattern word */
813     for (i = 0; i < 3; ++i)
814         *mem++ = patptr->mem_bank[i];
815     return(SUCCESS);
816 }
817
818 pel_build_pattern_control(first_pattern_no, pattern_count, mode)
819 {
820     if (!Host) return SUCCESS;
821     return build_pattern_control(first_pattern_no, pattern_count, mode);
822 }
823
824
825 char *Soft_patterns[2] = {
826     "z2f5f5z5c5c5c5c5c5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z5z
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pe_l_pattern.c	DATE 5/23/89	PAGE # 8/137
LINE #		SOURCE TEXT		
841				
842				
843				
844				
845				
846				
847				
848				
849				
850				
851				
852				
853				
854				
855				
856				
857				
858				
859				
860				
861				
862				
863				
864				
865				
866				
867				
868				
869				
870				
871				
872				
873				
874				
875				
876				
877				
878				
879				
880				
881				
882				
883				
884				
885				
886				
887				
888				
889				
890				
891				
892				
893				
894				
895				
896				
897				
898				
899				
900				
901				
902				
903				
904				
905				
906				
907				
908				
909				
910				
911				
912				
913				
914				
915				
916				
917				
918				
919				
920				
921				
922				
923				
924				
925				
926				
927				
928				
929				
930				
931				
932				
933				
934				
935				
936				
937				
938				
939				
940				
941				
942				
943				
944				
945				
946				
947				
948				
949				
950				
951				
952				
953				
954				
955				
956				
957				
958				
959				
960				

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_pattern.c

DATE

5/23/89

PAGE #

TIME

4:41:28 pm

9/138

```

LINE #          SOURCE TEXT
961      ln_message("VSE = 45.2F", vsh);
962      }
963      ln_message("  DUTVCC = 45.2F #####\n", dutvcc);
964
965      for (bit = 8; bit < 12; bit++) {
966          if (find_voltage(bit, &voltage, &dutvcc) == FAILURE) return(FAILURE);
967          if (value == direction) {
968              current = (dutvcc - voltage) / Resistance(bit) * 1000.0;
969          } else {
970              current = voltage / Resistance(bit) * 1000.0;
971          }
972          ln_message("bit %2d, voltage = 45.2F, current = 45.2F, resistance = 44.1F\n",
973              bit, voltage, current, Resistance(bit) / 1000.0);
974      }
975      return(SUCCESS);
976  }
977
978
979
980
981  pel_eval_soft_drivers()
982  {
983      register PEL *pel
984          = (PEL *) (pel_addr(current_lane, current_pel));
985      int magic, value, soft_drivers;
986
987      if (diag_clear_errors() != SUCCESS)
988          return(FAILURE);
989
990      if (pel_diag_dab_test_init(PUBLIC_DAB) != SUCCESS) {
991          (void) pel_errout("Unable to initialize PEL/DAB for Soft-Driver evaluation.\n");
992          return(FAILURE);
993      }
994
995      pel->car.bit.eeprom_sel = 1;          /* TURN ON resistors on DIAGDAB */
996
997      for (magic = 0; magic < 5; magic++) {
998          ln_message("\n***** Testing MAGIC[%x] *****\n", magic);
999          for (value = 0; value < 2; value++) {
1000              for (soft_drivers = 1; soft_drivers < 16; soft_drivers++) {
1001                  ln_message("Testing % Soft-Drivers OutX\n", soft_drivers);
1002                  if (load_drive_soft(magic, value, soft_drivers) == FAILURE) return(FAILURE);
1003                  if (eval_current(value, FORWARD, 0.0, 0.0) == FAILURE) return(FAILURE);
1004                  if (eval_current(value, REVERSE, 0.4, 2.9) == FAILURE) return(FAILURE);
1005              }
1006          }
1007      }
1008      return(SUCCESS);
1009  }

```

Copyright 1989  
Logic Modeling Systems

HEADER FILE  
diags/pel\_preamble.h

DATE	5/23/89	PAGE #
TIME	4:41:29 pm	1/139

LINE #	HEADER TEXT
1	/* SCCS ID: pel_preamble.h rev 3.1, 4/24/89 at 07:50:22 */
2	
3	typedef struct {
4	/* during the first real pattern */
5	unsigned :4,
6	:1, /* needed because MC will not turn on properly */
7	hdns:1,
8	:2, /* needed to initialize EDGE/NDGE */
9	seqhd:1,
10	seqnd:1,
11	lrychd:1,
12	lrycnd:1,
13	hdns:1,
14	doff:1,
15	set:2,
16	reset:2,
17	ctl_out:1,
18	load_adns:1, /* used to load the EDEN/FORMAT registers */
19	adns:4,
20	toggle:1,
21	format:2,
22	select_pel:1,
23	} PEL_PREAMBLE;
24	
25	typedef union {
26	PEL_PREAMBLE b;
27	unsigned long x;
28	} preamble;
29	
30	#define PREAMBLE_SIZE 28
31	

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_sst.c

DATE  
5/23/89

PAGE #  
1/140

TIME  
4:41:29 pm

LINE #

SOURCE TEXT

1

/\* SCCS ID: pel\_sst.c rev 3.1, 4/24/89 at 07:50:25 \*/

2

3

#include "common.h"

4

#include "mod\_def.h"

5

#include "moduler\_exta.h"

6

#include "vrtx.h"

7

#include "ls\_diags.h"

8

#include "tmg.h"

9

#include "tmg\_exta.h"

10

#include "tmg\_def.h"

11

#include "pac.h"

12

#include "pac\_def.h"

13

#include "pac\_exta.h"

14

#include "magic.h"

15

#include "pel.h"

16

#include "pel\_preamble.h"

17

18

long pel\_sst\_debug = 0;

19

20

/\* short sensor delay is spec'd at 200 ns minimum. \*/

21

22

#ifdef info

23

24

/\*

25

The way we test this puppy is to create a preamble similar to

26

the one used in the circ test, but having all pins using the RT

27

clock, all hard and medium drivers on, all data bits going the

28

same way.

29

30

Then I change one pin and see that the short sensors

31

don't come on after a short pulse (200ns) and that they do

32

come on after a relatively long time (1 us).

33

34

Anyway, I walk a 1 through all of the data pins to verify

35

that the sensors trip when they should (1us pulses) and

36

don't when they're not (200ns pulses).

37

38

Then I walk a zero across the pins.

39

40

I then move the edges around to find where the short

41

sensor trips.

42

\*/

43

#endif info

44

45

pel\_short\_sensor\_test()

46

{

47

long byte, trip, walking\_zeroes, retval = SUCCESS;

48

register char \*pattern\_string;

49

static long edgetime[6] =

50

200000, 00000, 000000, 00000, 00000, 0

51

;

52

if (diag\_clear\_errors() != SUCCESS)

53

return(FAILURE);

54

if (pel\_lane\_init() != SUCCESS) { /\* set up pac/tmg/pel/dab \*/

55

(void)lm\_error("Unable to initialize lane to\n", current\_lane + 'A');

56

return(FAILURE);

57

if (pel\_dab\_init(PUBLIC\_DAB) != SUCCESS) {

58

(void)lm\_error("Unable to initialize DAB to\n", PEL\_dab + 'A');

59

return(FAILURE);

60

if (set\_vlog1(0.8) == FAILURE) return(FAILURE);

61

if (set\_vlogh(2.0) == FAILURE) return(FAILURE);

62

if (set\_vlth(0.3) == FAILURE) return(FAILURE);

63

if (set\_val(0.4) == FAILURE) return(FAILURE);

64

if (set\_vah(4.0) == FAILURE) return(FAILURE);

65

if (set\_vthh(4.4) == FAILURE) return(FAILURE);

66

lm\_message("Playing patterns");

67

for (walking\_zeroes = 0; walking\_zeroes < 2; ++walking\_zeroes) {

68

for (trip = 0; trip < 2; ++trip) {

69

lm\_message("  ");

70

if (trip) {

71

/\* long pulse should make sensor trip \*/

72

edgetime[0] = 100000;

73

} else {

74

/\* short pulse should not make sensor trip \*/

75

edgetime[0] = 200000;

76

}

77

if (tmg\_set\_timing(15000001, edgetime, 01, 1500001, 25000001)

78

!= SUCCESS) {

79

return FAILURE;

80

}

81

for (byte = 0; byte < 2; ++byte) {

82

if (walking\_zeroes) {

83

pattern\_string = byte?

84

"x1f5x5x5f5f5f5x5

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/pel\_sst.c

DATE 5/23/89

PAGE #

TIME 4:41:29 pm

2/141

```

121 if (pel_short_sensor_measure() != SUCCESS) {
122     pel_error("Short sensor measure failed.\n");
123     retval = FAILURE;
124 }
125 return retval;
126 }
127
128 pel_short_bytes(byte, trip)
129 {
130     int retval = SUCCESS, time_out, play_status;
131     register long chip, bit, pin;
132     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
133
134     /* long pulse should make sensor trip */
135     if ((time_out = pec_play_play()) == 0)
136     {
137         (void) pel_error("Unable to prepare for play.\n");
138         return(FAILURE);
139     }
140
141     for (chip = 0; chip < 5; ++chip) {
142         for (bit = 0; bit < 8; ++bit) {
143             pin = chip * 16 + byte * 8 + bit;
144             toggle_pattern_bit(22, pin); /* turn off SEQMD for shorted pin */
145             toggle_pattern_bit(28, pin);
146
147             /* clear pel errors */
148             pel_clear_pel_errors();
149
150             /* disable backplane errors */
151             pel->car.bit.magic_error_enable = 1;
152
153             play_status = pec_play(time_out);
154             if ((pel_sst_delay & 1))
155                 use_play_till_busy();
156             if (play_status == FAILURE) {
157                 (void) ls_error("Lane %d Pel %d: Play error trip %d byte %d chip %d bit %d.\n",
158                     current_lane + 'A', current_pel,
159                     trip, byte, chip, bit);
160                 pec_play_cleanup();
161                 return(FAILURE);
162             }
163
164             if (report_by_error()) {
165                 (void) ls_error("Lane %d Pel %d: Unexpected backplane error from pin %d play.\n",
166                     current_lane + 'A', current_pel,
167                     pin);
168                 return(FAILURE);
169             }
170             if (check_short_status(pin, trip) != SUCCESS) {
171                 retval = FAILURE;
172                 if (ls_error("Lane %d Pel %d Chip %d: Pin %d short sensor %s.\n",
173                     current_lane + 'A', current_pel, chip,
174                     pin,
175                     trip? "did not trip" : "tripped") != SUCCESS)
176                     return FAILURE;
177                 toggle_pattern_bit(22, pin); /* turn back on SEQMD after shorting */
178                 toggle_pattern_bit(28, pin);
179             }
180         }
181     }
182     return retval;
183 }
184
185 set_pattern_bit(pattern, pin, value)
186 {
187     /*
188     Bank 0      Bank 1      Bank 2
189     +-----+ +-----+ +-----+
190     | 31 | 0 31 | 0 31 | 0 31 |
191     +-----+ +-----+ +-----+
192     | pin 79 | ..... | pin 0 | control |
193     +-----+ +-----+ +-----+
194     */
195     PAT_WORD patptr;
196     register long bank = 2 - ((pin + 16) >> 5);
197     register long mask = 1 << ((pin + 16) & 0x1f);
198     (void) pec_read_pattern(pattern, &patptr);
199     patptr.mem_bank[bank] = (value & 1)?
200     patptr.mem_bank[bank] & mask;
201     patptr.mem_bank[bank] & ~mask;
202     (void) pec_write_pattern(pattern, &patptr);
203 }
204
205 toggle_pattern_bit(pattern, pin)
206 {
207     /*
208     Bank 0      Bank 1      Bank 2
209     +-----+ +-----+ +-----+
210     | 31 | 0 31 | 0 31 | 0 31 |
211     +-----+ +-----+ +-----+
212     | pin 79 | ..... | pin 0 | control |
213     +-----+ +-----+ +-----+
214     */
215     PAT_WORD patptr;
216     register long bank = 2 - ((pin + 16) >> 5);
217     register long mask = 1 << ((pin + 16) & 0x1f);
218     register long value;
219     (void) pec_read_pattern(pattern, &patptr);
220     value = ((patptr.mem_bank[bank] & mask) == 0);
221     patptr.mem_bank[bank] = (value & 1)?
222     patptr.mem_bank[bank] & mask;
223     patptr.mem_bank[bank] & ~mask;
224     (void) pec_write_pattern(pattern, &patptr);
225 }
226
227 check_short_status(pin, trip)
228 {
229     register long retval;
230     register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
231
232     if (pel->car.bit.magic_error == 0) {
233         retval = identify_shorted_pin(trip? pin: -1); /* short sample register */
234     }
235 }

```



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_sst.cDATE 5/23/89  
TIME 4:41:29 pmPAGE #  
3/142

```

LINE #          SOURCE TEXT
241      } else retval = trip? FAILURE : SUCCESS;
242
243      /* clear pel errors */
244      pel_clear_pel_errors();
245
246      /* enable backplane errors */
247      pel->car.bit.magic_error_enable = 0;
248
249      return retval;
250
251  }
252
253  /* identify_shorted_pin returns SUCCESS iff the one pin tripped */
254  identify_shorted_pin(pis)
255  {
256      register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
257      register long chip, pischip, bit;
258      short word, pisword, error;
259      long error_pin, retval = SUCCESS;
260
261      if (pel->ast.debug & 4) {
262          for (chip = 0; chip < 5; ++chip) {
263              lm_message("0x41 - ", pel->magic_chip[chip].reg[8] & 0xffff);
264              lm_message("\n");
265          }
266      }
267
268      if (pis != -1) {
269          pischip = pis >> 4;
270          pisword = (1 << (pis & 0xf));
271      } else {
272          pischip = 0;
273          pisword = 0;
274      }
275
276      for (chip = 0; chip < 5; ++chip) {
277          word = pel->magic_chip[chip].reg[8];
278          if (!(error = (chip == pischip)? word - pisword : word))
279              continue;
280          retval = FAILURE;
281          for (bit = 0; bit < 16; ++bit) {
282              if (error & (1 << bit)) {
283                  error_pin = bit + 16 * chip;
284                  if (lm_error("Lane %c Pel %d Pin %d (chip %d bit %d) shorted on pin %d test.\n",
285                      current_lane + 'A', current_pel,
286                      error_pin,
287                      chip, bit,
288                      pis) == FAILURE)
289                      return FAILURE;
290              } else {
291                  if (lm_error("Lane %c Pel %d Pin %d (chip %d bit %d) did not trip.\n",
292                      current_lane + 'A', current_pel,
293                      error_pin, chip, bit) == FAILURE)
294                      return FAILURE;
295              }
296          }
297      }
298
299      return retval;
300  }
301
302  pel_reset_all_pels_in_lane()
303  {
304      register PEL *pel;
305      register int pelno;
306
307      for (pelno=0; pelno < NUMBER_OF_PELS; ++pelno) {
308          if (probe_pel(current_lane, pelno) == SUCCESS) {
309              pel = (PEL *) (pel_addr(current_lane, pelno));
310              pel->car.bit.resetL = 0;
311          }
312      }
313      lm_message("\n");
314  }
315
316  pel_clear_pel_errors()
317  {
318      short dummy;
319      register PEL *pel = (PEL *) (pel_addr(current_lane, current_pel));
320      /* clear pel errors */
321
322      dummy = pel->magic_chip[0].reg[15];
323      pel->car.bit.resetL = 0;
324      pel->car.bit.resetL = 1;
325      return (int)dummy; /* about 11msec */
326  }
327
328  /* the following functions are used to measure short trip time */
329  pel_short_sensor_measure()
330  {
331      long byte, timeout, retval = SUCCESS;
332      register char *pattern_string;
333      long chip, bit, pin, time;
334      long min_time, max_time;
335
336      if (pel_lane_init() != SUCCESS) { /* set up pec/tmg/pel/dab */
337          (void)lm_error("Unable to initialize lane %c\n", current_lane + 'A');
338          return(FAILURE);
339      }
340
341      if (pel_dab_init(PUBLIC_DAB) != SUCCESS) {
342          (void)lm_error("Unable to initialize DAB at Lane %c, PEL %d\n",
343              current_lane + 'A', current_pel);
344          return(FAILURE);
345      }
346
347      if (set_vlog1(0.8) == FAILURE) return(FAILURE);
348      if (set_vlog2(2.0) == FAILURE) return(FAILURE);
349      if (set_vlth(0.3) == FAILURE) return(FAILURE);
350      if (set_val(0.4) == FAILURE) return(FAILURE);
351      if (set_vsh(4.0) == FAILURE) return(FAILURE);
352      if (set_vth(4.4) == FAILURE) return(FAILURE);
353
354      min_time = 10000000; /* impossibly large */
355      max_time = 0; /* impossibly small */
356      lm_message("Measuring trip times");
357      for (byte = 0; byte < 2; ++byte) {
358          pattern_string = byte?
359      }

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_util.c	DATE 5/23/89	PAGE # 1/144
LINE #	SOURCE TEXT			
1	/* SCOS_ID: pel_util.c rev 3.1, 4/24/89 at 07:50:38 */			
2	/*			
3	.....			
4	* PEL utilities			
5	* used in PEL diag routines			
6	* .....			
7	* .....			
8	* .....			
9	* .....			
10	#include <math.h>			
11	#include "common.h"			
12	#include "mod_def.h"			
13	#include "modeler_ext.h"			
14	#include "wrx.h"			
15	#include "lm_diags.h"			
16	#include "tag.h"			
17	#include "tag_ext.h"			
18	#include "tag_def.h"			
19	#include "pac.h"			
20	#include "pac_def.h"			
21	#include "pac_ext.h"			
22	#include "magic.h"			
23	#include "pel.h"			
24	.....			
25	/* void flush_key_buf()			
26	* .....			
27	* INPUT: none			
28	* OUTPUT: none			
29	* DESCRIPTION: Clears key buffer.			
30	* .....			
31	flush_key_buf()			
32	{			
33	while(lm_check_key() != 0)			
34	lm_get_key();			
35	}			
36	.....			
37	/* pel_tag_init(void)			
38	* .....			
39	* This function initializes the Timing Generator to default			
40	* settings.			
41	* .....			
42	* Inputs: none			
43	* Outputs: function returns SUCCESS or FAILURE			
44	* .....			
45	pel_tag_init()			
46	{			
47	long edgetime(6);			
48	.....			
49	lm_message("Initializing Timing Generator to Defaults\n");			
50	.....			
51	if (tag_restore_calibration() != SUCCESS) {			
52	return FAILURE;			
53	}			
54	/*			
55	* These timings are absolutely critical to a number of different tests. They			
56	* should never get changed!!!			
57	* .....			
58	edgetime(0) = 5000;			
59	edgetime(1) = 5000;			
60	edgetime(2) = 5000;			
61	edgetime(3) = 15000;			
62	edgetime(4) = 5000;			
63	edgetime(5) = 0;			
64	.....			
65	if (tag_set_timing(40000L, edgetime, 0L, 100000L, 8000000L) != SUCCESS) {			
66	return FAILURE;			
67	}			
68	.....			
69	lane_select(lane_code(current_lane));			
70	.....			
71	lm_delay(1); /* wait for PEL to lock */			
72	return(SUCCESS);			
73	}			
74	.....			
75	/* pel_pac_init(void)			
76	* .....			
77	* This function initializes the PAC			
78	* .....			
79	* Inputs: none			
80	* Outputs: function returns SUCCESS or FAILURE			
81	* .....			
82	pel_pac_init()			
83	{			
84	if (pac[current_lane].exists == TRUE) {			
85	if (pac_stack_pane(current_lane) != SUCCESS) {			
86	(void)pel_error("Unable to set up pattern memory.\n");			
87	return(FAILURE);			
88	}			
89	} else {			
90	(void)pel_error("There is no Pattern Controller in this lane.\n");			
91	return(FAILURE);			
92	}			
93	.....			
94	Clear_pac_errors(current_lane);			
95	return(SUCCESS);			
96	}			
97	.....			
98	/* pel_dab_init()			
99	* .....			
100	* This function initializes the DAB			
101	* .....			
102	* Inputs: none			
103	* Outputs: function returns SUCCESS or FAILURE			
104	* .....			
105	/*			
106	pel_dab_init(mode)			
107	{			
108	int mode;			
109	{			
110	register PEL *pel			
111	= (PEL *) (pel_addr(current_lane, current_pel));			
112	u_short dummy;			
113	.....			
114	dummy = pel->magic_chip[0].reg[15]; /* reset MAGIC chip errors */			
115	.....			
116	pel->car.bit.resetL = 0; /* reset the PEL */			
117	pel->car.bit.initialize = 0;			
118	pel->car.bit.eeprom_in = 0;			
119	pel->car.bit.eeprom_clk = 0;			
120	pel->car.bit.eeprom_sel = 0;			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/pel_util.c	DATE 5/23/89 TIME 4:41:30 pm	PAGE # 2/145
LINE #	SOURCE TEXT			
121	pel->car.bit.magic_error_casual = 0;			
122	pel->car.bit.is_woe_led = 1;			
123	pel->car.bit.initialize = 1;			
124	pel->car.bit.reset1 = 1;			
125	if (mode == PUBLIC_DAB) {			
126	pel->car.bit.private = 0;			
127	if (pel->car.reg != 0x1f13) {			
128	(void)pel_error("Unable to initialize PUBLIC DAB\n");			
129	return(FAILURE);			
130	}			
131	} else { /* mode == PRIVATE */			
132	pel->car.bit.private = 1;			
133	if (pel->car.reg != 0x1f17) {			
134	(void)pel_error("Unable to initialize PRIVATE DAB\n");			
135	return(FAILURE);			
136	}			
137	}			
138	return(SUCCESS);			
139	#ifdef lint			
140	if (dummy), /* shut lint up */			
141	#endif lint			
142	return(SUCCESS);			
143	}			
144	}			
145	}			
146	}			
147	}			
148	}			
149	/* pel_lase_init(void)			
150	/* This function initializes the lase (and the current PEL)			
151	/* Inputs: none			
152	/* Outputs: function returns SUCCESS or FAILURE			
153	*/			
154	pel_lase_init()			
155	{			
156	/* Clear any PEL errors which may be present */			
157	pel_clear_pel_errors();			
158	if (pel_tmg_init() != SUCCESS) {			
159	(void)pel_error("Unable to initialize TMC\n");			
160	return(FAILURE);			
161	}			
162	if (pel_pac_init() != SUCCESS) {			
163	(void)pel_error("Unable to initialize PAC\n");			
164	return(FAILURE);			
165	}			
166	return(SUCCESS);			
167	}			
168	}			
169	more(st, size, count)			
170	{			
171	register long illegal;			
172	static char sol[] = "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"; /* start of line */			
173	/* message: "More--(addr)", 100*(1+st)/(1+size);			
174	do {			
175	illegal = 0;			
176	switch (lm_get_key()) {			
177	case 'G': lm_message("table\n", sol, ""); return 0; break;			
178	case 'A': --count; lm_message("to", sol); break;			
179	case 'I': lm_message("is", sol); break;			
180	default: lm_message("007", ++illegal, /* illegal char */			
181	}			
182	Clear key buf();			
183	while (illegal);			
184	return count;			
185	}			
186	}			
187	}			
188	}			
189	}			
190	}			
191	}			
192	#define is_space(c) (((c) == ' ')    ((c) == '\t'))			
193	#define is_num(c) (((c) >= '0') && ((c) <= '9'))			
194	char *			
195	get_range(s, from, to)			
196	register char *s;			
197	register u_long *from, *to;			
198	{			
199	char buffer[128];			
200	register char *b;			
201	while (is_space(*s)) ++s;			
202	b = buffer;			
203	if (!is_num(*s)) return 0; /* syntax error */			
204	while (is_num(*s)) *b++ = *s++;			
205	*b = '\0';			
206	*from = *to = atoi(buffer);			
207	while (is_space(*s)) ++s;			
208	if (*s == '-') {			
209	/* Range of values */			
210	++s;			
211	while (is_space(*s)) ++s;			
212	b = buffer;			
213	if (!is_num(*s)) return 0; /* syntax error */			
214	while (is_num(*s)) *b++ = *s++;			
215	*b = '\0';			
216	*to = atoi(buffer);			
217	while (is_space(*s)) ++s;			
218	}			
219	if (*s == ',') ++s; /* optional comma separator */			
220	return s;			
221	}			
222	}			
223	}			
224	pel_set_timing()			
225	{			
226	static long edgetime[6] = {			
227	10000, 10000, 10000, 10000, 10000, 0			
228	};			
229	static long period = 40000;			
230	long edge;			
231	char buffer[80];			
232	diag_get_long(&period, "Period", 40000, 0x7fffffff);			
233	for (edge = 0; edge < 6; ++edge) {			
234	sprintf(buffer, "Edge %d", edge);			
235	}			
236	}			
237	}			
238	}			
239	}			
240	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/pel\_util.c

DATE 5/23/89  
TIME 4:41:30 pm

PAGE #  
3/146

```

LINE #          SOURCE TEXT
241      diag_get_logg((edgetime[edge]), buffer, 01, period);
242      }
243      if (tmg_set_timing(period, edgetime, 01, 1000001, 25000001) != SUCCESS) {
244          return FAILURE;
245      }
246      return SUCCESS;
247  }
248  }
249  pel_set_voltages()
250  {
251      long vlogl = 800;
252      long vlogh = 2000;
253      long vlth = 300;
254      long val = 400;
255      long vsh = 4000;
256      long vlth = 4400;
257
258      do {
259          diag_get_logg(&vlogl, "vlogl (mv)", 01, 120001);
260          } while (set_vlogl(0.001 * (float)(vlogl)) != SUCCESS);
261      do {
262          diag_get_logg(&vlogh, "vlogh (mv)", 01, 120001);
263          } while (set_vlogh(0.001 * (float)(vlogh)) != SUCCESS);
264      do {
265          diag_get_logg(&vlth, "vlth (mv)", 01, 120001);
266          } while (set_vlth(0.001 * (float)(vlth)) != SUCCESS);
267      do {
268          diag_get_logg(&val, "val (mv)", 01, 120001);
269          } while (set_val(0.001 * (float)(val)) != SUCCESS);
270      do {
271          diag_get_logg(&vsh, "vsh (mv)", 01, 120001);
272          } while (set_vsh(0.001 * (float)(vsh)) != SUCCESS);
273      do {
274          diag_get_logg(&vlth, "vlth (mv)", 01, 120001);
275          } while (set_vlth(0.001 * (float)(vlth)) != SUCCESS);
276      }
277  }
278  pel_loop_play()
279  {
280      int time_out;
281      flush_key_buf();
282
283      if ((time_out = pac_pro_play()) == 0)
284      {
285          (void)pel_error("Unable to prepare for looping play.\n");
286          return(FAILURE);
287      }
288      in_message("Hit key to stop...");
289
290      while (!in_check_key()) {
291          if (pac_play(time_out) == FAILURE) {
292              (void)pel_error("pel_play_patterns_string failed in pac_play.\n");
293              pac_play_cleanup();
294              break;
295          }
296      }
297      flush_key_buf();
298      return(SUCCESS);
299  }
300
301

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/probe.c	DATE 5/23/89 TIME 4:41:30 pm	PAGE # 1/147
LINE #	SOURCE TEXT			
1	/* SCSS ID: probe.c rev 3.1, 4/24/89 at 07:50:31 */			
2	.....			
3	probe.c			
4	.....			
5	/* Probe function			
6	used in diagnostics			
7	.....			
8	.....			
9	.....			
10	.....			
11	#include <math.h>			
12	#include "common.h"			
13	#include "lm_diags.h"			
14	#include "mod_def.h"			
15	#include "vrtx.h"			
16	#include "tmg.h"			
17	#include "pac.h"			
18	#include "magic.h"			
19	#include "pel.h"			
20	.....			
21	int			
22	probe_tmg()			
23	{			
24	if(lm_read_probe((long)(0x80000000)) != SUCCESS)			
25	{			
26	(void)lm_warning("No Timing Generator installed.\n");			
27	return(FAILURE);			
28	}			
29	else			
30	return(SUCCESS);			
31	}			
32	.....			
33	int			
34	probe_all_lanes()			
35	{			
36	int lane_no;			
37	.....			
38	/* Reset backplane lanes */			
39	if(diag_tmg_reset(TRUE) != SUCCESS)			
40	{			
41	(void)lm_error("Could not reset backplane during lane probe.\n");			
42	return(FAILURE);			
43	}			
44	for(lane_no = 0; lane_no < NUMBER_OF_LANES; lane_no++)			
45	{			
46	if(probe_lane(lane_no) == SUCCESS)			
47	return(SUCCESS);			
48	}			
49	(void)lm_warning("There are no PACs or PELS installed.\n");			
50	return(FAILURE);			
51	}			
52	.....			
53	int			
54	probe_lane(lane_no)			
55	{			
56	int lane_no;			
57	.....			
58	if((probe_pac(lane_no) == SUCCESS)    (probe_all_pels(lane_no) == SUCCESS))			
59	return(SUCCESS);			
60	else			
61	return(FAILURE);			
62	}			
63	.....			
64	int			
65	probe_pac(lane_no)			
66	{			
67	int lane_no;			
68	.....			
69	if(lm_read_probe((long)(LANE_A_OFFSET + (lane_no * LANE_SIZE) +			
70	PAC_REC_OFFSET)) != SUCCESS)			
71	return(FAILURE);			
72	else			
73	return(SUCCESS);			
74	}			
75	.....			
76	int			
77	probe_all_pels(lane_no)			
78	{			
79	int lane_no;			
80	.....			
81	int slot_no;			
82	for(slot_no = 0; slot_no < NUMBER_OF_SLOTS; slot_no++)			
83	{			
84	if(probe_pel(lane_no, slot_no) == SUCCESS)			
85	return(SUCCESS);			
86	}			
87	return(FAILURE);			
88	}			
89	.....			
90	int			
91	probe_pel(lane_no, slot_no)			
92	{			
93	int lane_no;			
94	int slot_no;			
95	.....			
96	if(lm_read_probe((long)(pel_addr(lane_no, slot_no))) != SUCCESS)			
97	return(FAILURE);			
98	else			
99	return(SUCCESS);			
100	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 1/148
LINE #	SOURCE TEXT			
1	/* SCCS ID: tmg_cal.c rev 3.2, 5/9/89 at 15:54:16 */			
2	/*-----*/			
3	/*			
4	Calib routines for the Timing Generator.			
5	/*-----*/			
6	*/			
7	*/			
8	*/			
9	#include <math.h>			
10	#include "common.h"			
11	#include "mod_def.h"			
12	#include "modeler_extn.h"			
13	#include "vrtx.h"			
14	#include "tmg.h"			
15	#include "tmg_def.h"			
16	#include "tmg_extn.h"			
17	#include "tmg_run.h"			
18	#include "pac.h"			
19	#include "pac_def.h"			
20	#include "pac_extn.h"			
21	#ifdef DIAGS			
22	#include "diag_setjmp.h"			
23	#endif DIAGS			
24	#include "istr.h"			
25	*/			
26	#ifdef DIAGS			
27	#define lm_message printf			
28	#define lm_warning printf			
29	#define lm_error printf			
30	null function({})			
31	#endif DIAGS			
32	*/			
33	#ifdef HOST			
34	#include "lm_diags.h"			
35	#define lm_message lm_message			
36	#define lm_error (++total_errors[current_depth], lm_message)			
37	#define lm_warning lm_message			
38	static long current_depth = 0;			
39	#endif HOST			
40	*/			
41	#undef MIN_PERIOD			
42	#define MIN_PERIOD 33333			
43	#define BIN_LIMIT 100 /* binary search count limit */			
44	*/			
45	static void tmg_assume_mint(),			
46	static long tmg_lowest_delay_edge();			
47	*/			
48	long tmg_cal_debug = 0;			
49	long tmg_reduction_factor = 82; /* % reduction for edge search */			
50	*/			
51	#define tmg_detect_always_high(x, e, p, t, c, actual) \			
52	tmg_detect_edge(x, e, DETECT_BOOL, \			
53	p, t, c, actual)			
54	#define tmg_detect_always_low(x, e, p, t, c, actual) \			
55	tmg_detect_edge(x, e, DETECT_BOOL DETECT_LOW, \			
56	p, t, c, actual)			
57	#define tmg_detect_sample_always_high(xr, sr, p, t7, ts, c, actual) \			
58	tmg_detect_sample(xr, sr, DETECT_BOOL DETECT_SAMPLE, \			
59	p, t7, ts, c, actual)			
60	#define tmg_detect_sample_always_low(xr, sr, p, t7, ts, c, actual) \			
61	tmg_detect_sample(xr, sr, \			
62	DETECT_BOOL DETECT_LOW DETECT_SAMPLE, \			
63	p, t7, ts, c, actual)			
64	#define tmg_g_t_ramp() (tmgptr->sample_delay_range)			
65	#define tmg_set_ramp(xramp) (tmgptr->sample_delay_range = (xramp))			
66	*/			
67	#define CALIBRATION_DONE DEADBEEF			
68	/* XXXXX */			
69	#undef tmg_set_threshold			
70	*/			
71	#define tmg_set_threshold(edge, threshold)			
72	{			
73	int edge, threshold;			
74	{			
75	int i, other_thresh;			
76	if(threshold < 128)			
77	other_thresh = 255;			
78	else			
79	other_thresh = 10;			
80	tmgptr->edge_delay[edge].delay = threshold;			
81	for(i = 0; i < 6; i++)			
82	{			
83	if(i == edge)			
84	continue;			
85	tmgptr->edge_delay[i].delay = other_thresh;			
86	}			
87	}			
88	tmg_calibrate()			
89	{			
90	return tmg_do_calibrate(1);			
91	}			
92	tmg_recalibrate()			
93	{			
94	return tmg_do_calibrate(0);			
95	}			
96	tmg_do_calibrate(destructive)			
97	{			
98	#ifdef info			
99	1. reset the clock board			
100	2. find the minimum threshold of the fast ramp for all edges			
101	3. set min_thresh of ramp to ramp 0's value for all edges;			
102	4. calculate slope & offset of fast ramp (0) for each edge			
103	5. set the delay line			
104	6. Finally calibrate all edges of all ramps			
105	7. Next measure edge 7 and sample's mint			
106	8. Measure edge7 and sample ramps			
107	9. Measure sample offsets			
108	10. Measure Early mode sample offsets			
109	#endif info			
110	#endif			
111	extern struct CALIB default_calib;			
112	static char me[] = "tmg_calibrate";			
113	*/			
114	*/			
115	*/			
116	*/			
117	*/			
118	*/			
119	*/			
120	*/			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

2/149

```

121 register long ramp, aramp, aramp, first_aramp, last_aramp, edge;
122 long threshold, thresh7, thresha;
123 char buffer[80];
124
125 long errors = 0;
126
127 INTR_BEGIN
128
129 /* if destructive, we should do a backplane reset (diag_tmg_reset(TRUE); */
130 /* because a side effect of that is to do a par_info_init */
131 if(diag_tmg_reset(destructive ? TRUE : FALSE) != SUCCESS)
132 {
133     tmg_report_failure(me, "diag_tmg_reset");
134     ++errors;
135     goto cleanup;
136 }
137
138 /* find the minimum threshold of the fast ramp for all edges */
139 tmg_set_delay(14); /* delay = 6ns + 14ns */
140
141 tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
142
143 if (destructive) {
144     tmgptr->backplane_reset = 0; /* assert backplane reset */
145     lm_message("Measuring edge minimum thresholds");
146     for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
147         lm_message(".");
148         if (tmg_measure_minth(01, edge, athreshold) != SUCCESS) {
149             tmg_report_failure(me, "tmg_measure_minth");
150             ++errors;
151             if (!(tmg_cal_debug & 0x20000000))
152                 goto cleanup;
153         }
154         calib.EdgeMinThresh[0][edge] = threshold + START_DEAD_TIME_FUDGE;
155     }
156     lm_message("Done\n");
157 } else {
158     if (!tmg_is_calibrated()) {
159         for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
160             calib.EdgeMinThresh[0][edge] = default_calib.EdgeMinThresh[0][edge];
161         }
162     }
163 }
164
165 /* set min thresh of ramp to ramp 0's value for all edges */
166 tmg_set_minth(calib);
167
168 lm_message("Measuring edge ramp slopes");
169
170 for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
171     for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
172         lm_message(".");
173         if (tmg_measure_ramp(ramp, edge, calib, &errors) != SUCCESS) {
174             sprintf(buffer, "tmg_measure_ramp(%d,%d)", ramp, edge);
175             tmg_report_failure(me, buffer);
176             ++errors;
177             if (!(tmg_cal_debug & 0x20000000))
178                 goto cleanup;
179         }
180     }
181 }
182
183 lm_message("Done\n");
184 #ifdef DIAGS
185     if (tmg_cal_debug & 0x40000000)
186         tmg_play_til_key();
187 #endif
188
189 /* set the delay line */
190
191 if (tmg_calibrate_delay(calib) != SUCCESS) {
192     tmg_report_failure(me, "tmg_calibrate_delay");
193     ++errors;
194     if (!(tmg_cal_debug & 0x20000000))
195         goto cleanup;
196 }
197 #ifdef DIAGS
198     if (tmg_cal_debug & 0x40000000)
199         tmg_play_til_key();
200 #endif
201
202 /* remeasure the offsets */
203
204 lm_message("Measuring edge offsets");
205 for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
206     for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
207         if (tmg_measure_offset(ramp, edge, calib) != SUCCESS) {
208             tmg_report_failure(me, "tmg_measure_offset");
209             ++errors;
210             if (!(tmg_cal_debug & 0x20000000))
211                 goto cleanup;
212         }
213     }
214 }
215
216 lm_message("Done\n");
217 #ifdef DIAGS
218     if (tmg_cal_debug & 0x40000000)
219         tmg_play_til_key();
220 #endif
221
222 if (destructive) {
223     /* Next measure edge 7 and sample's minth */
224     /* (edge 7 is the edge that triggers a SAMPLE) */
225     lm_message("Measuring Edge7 and sample minimum thresholds");
226     ramp = 0;
227     tmg_set_sample_threshold(50); /* sure to cause a trigger */
228     tmg_set_edge7_threshold(50); /* sure to cause a trigger */
229     if (tmg_measure_edge7_sample_minth(01, 01, &thresh7, &thresha) != SUCCESS) {
230         tmg_report_failure(me, "tmg_measure_edge7_sample_minth");
231         ++errors;
232         if (!(tmg_cal_debug & 0x20000000))
233             goto cleanup;
234     }
235     lm_message("Done\n");
236 #ifdef DIAGS
237     if (tmg_cal_debug & 0x40000000)
238         tmg_play_til_key();
239 #endif
240     for (aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
241         calib.Edge7MinThresh[aramp] = thresh7 + START_DEAD_TIME_FUDGE;

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 3/150
LINE #	SOURCE TEXT			
241	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++)			
242	calib.SampleMinThresh[aramp] = threshs + START_DEAD_TIME_FUDGE;			
243	} else {			
244	if (!tmg_is_calibrated()) {			
245	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++)			
246	calib.Edge7MinThresh[aramp] = Default_calib.Edge7MinThresh[aramp];			
247	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++)			
248	calib.SampleMinThresh[aramp] = Default_calib.SampleMinThresh[aramp];			
249	}			
250	}			
251	lm_message("Measuring Sample ramp slopes");			
252	for (aramp = 0; aramp < NUMBER_OF_SRAMP5; ++aramp) {			
253	lm_message(".");			
254	if (tmg_measure_edge7_ramp(aramp, &calib) != SUCCESS) {			
255	tmg_report_failure(me, "tmg_measure_edge7_ramp");			
256	++errors;			
257	if (!(tmg_cal_debug & 0x20000000))			
258	goto cleanup;			
259	}			
260	else {			
261	switch (aramp) {			
262	case 0:			
263	first_aramp = 0; /* just do first one */			
264	last_aramp = 0;			
265	break;			
266	case 1:			
267	first_aramp = 1; /* do next 2 */			
268	last_aramp = 2;			
269	break;			
270	case 2:			
271	first_aramp = 3; /* do last one */			
272	last_aramp = 3;			
273	break;			
274	default:			
275	first_aramp = 100; /* don't do any */			
276	last_aramp = 0;			
277	break;			
278	}			
279	for(aramp = first_aramp; aramp <= last_aramp; aramp++) {			
280	lm_message(".");			
281	if (tmg_measure_sample_ramp(aramp, aramp, &calib, &errors)			
282	!= SUCCESS) {			
283	tmg_report_failure(me, "tmg_measure_sample_ramp");			
284	++errors;			
285	if (!(tmg_cal_debug & 0x20000000))			
286	goto cleanup;			
287	}			
288	}			
289	}			
290	lm_message("Done\n");			
291	#ifdef DIAGS			
292	if (tmg_cal_debug & 0x00000000)			
293	tmg_play_til_key();			
294	#endif DIAGS			
295	lm_message("Measuring sample offsets");			
296	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++) {			
297	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++) {			
298	lm_message(".");			
299	if (tmg_measure_sample_offset_only(aramp, aramp, &calib,			
300	EDGE7SAMPLETRIGGERMODE) != SUCCESS)			
301	{			
302	tmg_report_failure(me, "tmg_measure_sample_offset_only\			
303	, Edge 7 mode");			
304	++errors;			
305	if (!(tmg_cal_debug & 0x20000000))			
306	goto cleanup;			
307	}			
308	}			
309	}			
310	}			
311	}			
312	for(aramp = 0; aramp < NUMBER_OF_SRAMP5; aramp++)			
313	{			
314	if (tmg_measure_sample_offset_only(01.aramp, &calib,			
315	EARLYSAMPLETRIGGERMODE) != SUCCESS)			
316	{			
317	tmg_report_failure(me, "tmg_measure_sample_offset_only, early mode");			
318	++errors;			
319	if (!(tmg_cal_debug & 0x20000000))			
320	goto cleanup;			
321	}			
322	}			
323	lm_message(".");			
324	}			
325	}			
326	lm_message("Done\n");			
327	#ifdef DIAGS			
328	if (tmg_cal_debug & 0x00000000)			
329	tmg_play_til_key();			
330	#endif DIAGS			
331	if (tmg_complete_calib_structure(&calib) != SUCCESS) {			
332	tmg_report_failure(me, "tmg_complete_calib_structure");			
333	++errors;			
334	if (!(tmg_cal_debug & 0x20000000))			
335	goto cleanup;			
336	}			
337	if (tmg_cal_debug & 0x00000000)			
338	{			
339	if (tmg_print_calib_structure(&calib, != SUCCESS)			
340	{			
341	tmg_report_failure(me, "tmg_print_calib_structure");			
342	++errors;			
343	if (!(tmg_cal_debug & 0x20000000))			
344	goto cleanup;			
345	}			
346	}			
347	}			
348	if (calib.EarlySampleMinDelay[0] > (long)calib.EdgeMinDelay[0])			
349	{			
350	++errors;			
351	lm_error("EarlySampleMinDelay[0] = %d ps > EdgeMinDelay[0] = %d ps\n",			
352	calib.EarlySampleMinDelay[0], calib.EdgeMinDelay[0]);			
353	}			
354	}			
355	}			
356	if (tmg_cal_debug & 0x00000000)			
357	{			
358	if (tmg_print_data_logging_messages(&calib) != SUCCESS)			
359	{			
360	}			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.cDATE 5/23/89  
TIME 4:41:30 pmPAGE #  
4/151

```

LINE # SOURCE TEXT
361 tmg_report_failure(m, "tmg_print_data_logging_messages");
362 errors++;
363 if(!((tmg_cal_debug & 0x20000000)))
364     goto cleanup;
365 }
366
367 cleanup:
368
369 tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
370
371 tmg_set_test_mode(0);
372 if ((errors == 0) || ((tmg_cal_debug & 0x20000000) != 0)) {
373     tmg_save_calibration();
374     calib.CalCompleted = CALIBRATION_DONE;
375 }
376 if (destructive) {
377     (void)diag_tmg_reset(1); /* full reset */
378 }
379
380 INTR_GOT_OWE
381 tmg_set_test_mode(0);
382 calib.CalCompleted = 0; /* unsuccessful */
383 (void)diag_tmg_reset(0);
384 intr();
385 INTR_END
386
387 if (errors) {
388     calib.CalCompleted = 0; /* unsuccessful */
389     return(FAILURE);
390 }
391 return(SUCCESS);
392
393 /*
394 tmg_incremental_calibrate(part, destructiveFlag)
395 *
396 * Does a portion of what tmg_do_calibrate() does, selected by
397 * 'part'. Only does minimum threshold measurements if the
398 * 'destructiveFlag' is TRUE (normally only done at power up and
399 * at very long intervals, if ever again).
400 */
401 int tmg_incremental_calibrate(part, destructive)
402 int part, destructive;
403 {
404     static struct CALIB tmpcal;
405     int returncode = SUCCESS;
406     long threshold, thresh7, threshs, edge, aramp, errors;
407     int save_delay_line;
408
409     if((part >= 0) && (part < 6) && !destructive)
410     {
411         edge = part & 6;
412         tmgptr->backplane_reset = 0; /* assert backplane reset */
413         save_delay_line = tmg_get_delay();
414         tmg_set_delay(14);
415         if(tmg_measure_minth(01, edge, &threshold) != SUCCESS)
416             returncode = FAILURE;
417         else
418             for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
419                 calib.EdgeMinThresh[aramp][edge] = threshold + START_DEAD_TIME_FUDGE;
420         tmg_set_delay(save_delay_line);
421         tmgptr->backplane_reset = 1;
422         tmg_set_test_mode(0);
423         return returncode;
424     }
425     else if((part >= 6) && (part < 30)) /* measure edge/ramp slopes */
426     {
427         edge = part & 6;
428         aramp = (part - 6) / 6;
429         save_delay_line = tmg_get_delay();
430         tmg_set_delay(14);
431         if(tmg_measure_ramp(aramp, edge, &tmpcal, &errors) != SUCCESS)
432             returncode = FAILURE;
433         tmg_set_delay(save_delay_line);
434         tmg_set_test_mode(0);
435         return returncode;
436     }
437     else if(part == 30) /* check or set delay line */
438     {
439         save_delay_line = tmg_get_delay();
440         if(tmg_calibrate_delay(&tmpcal) != SUCCESS)
441             returncode = FAILURE;
442         if(!destructive)
443         {
444             if(abs((int)tmg_get_delay() - save_delay_line) > 1)
445                 returncode = FAILURE;
446             tmg_set_delay(save_delay_line);
447         }
448         tmg_set_test_mode(0);
449         return returncode;
450     }
451     else if((part > 30) && (part < 45)) /* measure edge/ramp offsets */
452     {
453         edge = (part - 37) & 6;
454         aramp = (part - 37) / 6;
455         if(tmg_measure_offset(aramp, edge, &tmpcal) != SUCCESS)
456             returncode = FAILURE;
457         tmg_set_test_mode(0);
458         return returncode;
459     }
460     else if((part == 45) && !destructive) /* measure Edge 7 min thresh */
461     {
462         if(tmg_measure_edge7_sample_minth(01, 01, &thresh7, &threshs) != SUCCESS)
463             returncode = FAILURE;
464         for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
465             tmpcal.Edge7MinThresh[aramp] = thresh7 + START_DEAD_TIME_FUDGE;
466         for(aramp = 0; aramp < NUMBER_OF_ERAMPS; aramp++)
467             tmpcal.SampleMinThresh[aramp] = threshs + START_DEAD_TIME_FUDGE;
468         tmg_set_test_mode(0);
469         return returncode;
470     }
471     else if((part > 45) && (part < 50)) /* measure Edge 7 slope */
472     {
473         aramp = (part - 46);
474         if(tmg_measure_edge7_ramp(aramp, &tmpcal) != SUCCESS)
475             returncode = FAILURE;
476     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

5/152

```

LINE #      SOURCE TEXT
481      tmg_set_test_mode(0);
482      return Returncode;
483  }
484  else if((part >= 50) && (part < 54)) /* measure sample slope */
485  {
486      aramp = (part - 50);
487      switch(aramp)
488      {
489          case 0: aramp = 0; break;
490          case 1: aramp = 1; break;
491          case 2: aramp = 1; break;
492          case 3: aramp = 2; break;
493      }
494      if(tmg_measure_sample_ramp(aramp, aramp, &stampcal, &errors) != SUCCESS)
495          Returncode = FAILURE;
496      tmg_set_test_mode(0);
497      return Returncode;
498  }
499  else if((part >= 54) && (part < 70))
500  {
501      aramp = (part - 54) % 4;
502      aramp = (part - 54) / 4;
503      if(tmg_measure_sample_offset_dely(aramp, aramp, &stampcal,
504      EDGE7SAMPLETRIGGERMODE) != SUCCESS)
505          Returncode = FAILURE;
506      tmg_set_test_mode(0);
507      return Returncode;
508  }
509  else if(part == 70)
510      return(tmg_process_and_transfer_calib_structure(&stampcal, &scalib));
511  else
512      return FAILURE;
513  }
514
515  tmg_calibrate_delay(cal)
516  register struct CALIB *cal;
517  {
518      static char me[] = "tmg_calibrate_delay";
519      long delta;
520      register long i;
521      register long longest_delay_edge, count;
522
523      lm_message("Setting delay line");
524
525      /* Now find the ramp 0 edge with the longest delay */
526      /* where delay = edgeoffset + (edgeminthresh * edgeofslope) */
527      longest_delay_edge = tmg_longest_delay_edge(0l, cal);
528
529      /* Set the aramp delay */
530      if (tmg_compute_delay(0l, longest_delay_edge, cal, &delta)
531      != SUCCESS) {
532          tmg_report_failure(me, "tmg_compute_delay");
533          return FAILURE;
534      }
535      if(tmg_cal_debug & 0x10000)
536          lm_message("as: adjusting delay by %d ps\n", me, delta);
537      tmg_adjust_delay(delta); /* do coarse adjustment */
538      if (tmg_compute_delay(0l, longest_delay_edge, cal, &delta)
539      != SUCCESS) {
540          tmg_report_failure(me, "tmg_compute_delay");
541          return FAILURE;
542      }
543      for (count = 0; i = (delta > 0), i -= (delta > 0), --count) {
544          if (delta == 0)
545              break;
546          delta = (delta > 0) ? 1000 : -1000;
547          tmg_adjust_delay(delta); /* do fine adjustment */
548          if (tmg_compute_delay(0l, longest_delay_edge, cal, &delta)
549          != SUCCESS) {
550              tmg_report_failure(me, "tmg_compute_delay");
551              return FAILURE;
552          }
553          if (count > 20) {
554              tmg_report_failure(me, "Delay adjustment");
555              return FAILURE;
556          }
557      }
558      lm_message("Done\n");
559      return SUCCESS;
560  }
561
562  /* tmg_assume_minth() assumes that the aramp 0 minth's have been measured
563  ** and that it is a good assumption that the other ramps minths are the
564  ** same
565  */
566
567  static void
568  tmg_assume_minth(cal)
569  register struct CALIB *cal;
570  {
571      register long threshold0, edge, ramp;
572
573      for (edge = 0; edge < NUMBER_OF_EDGES; ++edge) {
574          threshold0 = cal->EdgeMinThresh[0][edge];
575          for (ramp = 1; ramp < NUMBER_OF_RAMPS; ++ramp) {
576              /* set min thresh of ramp to ramp 0's value */
577              cal->EdgeMinThresh[ramp][edge] = threshold0;
578          }
579      }
580  }
581
582  /*
583  ** tmg_longest_delay_edge(ramp, cal)
584  ** returns the edge with the longest delay for a given ramp
585  ** where delay = offset + (minthresh * slope)
586  */
587
588  static long
589  tmg_longest_delay_edge(ramp, cal)
590  register long ramp;
591  register struct CALIB *cal;
592  {
593      register long edge, longest_delay_edge = 0;
594      register long delay, longest_delay;
595  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

6/153

```

LINE # SOURCE TEXT
601 longest_delay = cal->EdgeOffset[ramp][0]
602 + (long)(cal->EdgeInitThresh[ramp][0]
603 + cal->EdgeSlope[ramp][0]);
604 for (edge = 1; edge < NUMBER_OF_EDGES; ++edge) {
605     delay = cal->EdgeOffset[ramp][edge]
606     + (long)(cal->EdgeInitThresh[ramp][edge]
607     + cal->EdgeSlope[ramp][edge]);
608     if (delay > longest_delay) {
609         longest_delay = delay;
610         longest_delay_edge = edge;
611     }
612 }
613 return longest_delay_edge;
614 }
615
616 struct measure_points {
617     long low, mid, high;
618     long slope; /* (psaw/th) */
619 }measure_points[] = {
620     {104, 155, 250, 500},
621     {18, 135, 250, 2000},
622     {12, 125, 250, 10000},
623     {10, 125, 250, 50000},
624 };
625
626 struct measure_sample_points {
627     long low, mid, high;
628     long slope; /* (psaw/th) */
629 }measure_sample_points[] = {
630     {10, 125, 250, 500},
631     {10, 125, 250, 1000},
632     {10, 125, 250, 2000},
633     {10, 125, 250, 4000},
634 };
635 /* long measure_early_sample_points[] = {104, 54, 29, 16}; */
636 long measure_early_sample_points[] = {125, 65, 37, 22};
637
638 struct measure_edge7_points {
639     long low, mid, high;
640     long slope; /* (psaw/th) */
641 }measure_edge7_points[] = {
642     {104, 155, 250, 500},
643     {18, 135, 250, 2000},
644     {12, 125, 250, 10000},
645     {10, 125, 250, 50000},
646 };
647
648 /* t is threshold array, p is period array, for general use */
649 /* during calibration of all ramps */
650 static long p[NUMBER_OF_RAMPS][NUMBER_OF_EDGES + 1][7];
651 static long t[NUMBER_OF_RAMPS][NUMBER_OF_EDGES + 1][7];
652
653 tmg_measure_ramp(ramp, edge, cal, errors)
654 register long ramp, edge, *errors;
655 register struct CALIB *cal;
656 {
657     static char me[] = "tmg_measure_ramp";
658     long try_tl;
659     register long dt, dp, slope;
660     long sepoints, point, temperrors;
661     long *plist, *tlist, *sepoints;
662     register long count;
663
664     #ifdef DIAGS
665     if (tmg_cal_debug & 2)
666         lm_message("tmg_measure_ramp(td,td,x)\n", ramp, edge, cal);
667     #endif DIAGS
668     /*
669     ** 1. find a point above but near the minimum threshold and longer
670     ** than the magic ship dead time.
671     ** 2. find a point high on the ramp.
672     ** 3. Calculate the slope and offset of that line.
673     ** 3a. Use the line to predict several points to do a fine calibration.
674     ** 3b. Results of fine calibration are used for slope and offset.
675     ** 4. find a point approx midway between the first two points using
676     ** results of 3b.
677     ** 5. Compute a linearity value.
678     **
679     ** Slope will be stored as a long. The units are v/s (or uv/us).
680     */
681     temperrors = 0;
682     tmg_set_slot_count(2);
683
684     t[ramp][edge][0] = measure_points[ramp].low;
685     if (!p[ramp][edge][0] = tmg_find_period(ramp, edge,
686         (long)t[ramp][edge][0]))
687     {
688         tmg_report_failure(me,
689             "can't find low period: tmg_find_period");
690         return(FAILURE);
691     }
692
693     t[ramp][edge][2] = measure_points[ramp].high;
694     if (!p[ramp][edge][2] = tmg_find_period(ramp, edge,
695         (long)t[ramp][edge][2]))
696     {
697         tmg_report_failure(me,
698             "can't find high period: tmg_find_period");
699         return(FAILURE);
700     }
701
702     if ((dt = t[ramp][edge][2] - t[ramp][edge][0]) == 0) {
703         lm_error("infinite edge ramp slope detected\n");
704         return FAILURE;
705     }
706
707     if ((dp = p[ramp][edge][2] - p[ramp][edge][0]) == 0) {
708         lm_error("zero edge ramp slope detected\n");
709         return FAILURE;
710     }
711
712     /* slopes are normally 500, 2000, 10000, and 50000 */
713     slope = (dp/dt);
714     if ((slope < 500) || (slope > 50000)) {
715         lm_error("illegal edge ramp slope td ps/thr\n", slope);
716         return FAILURE;
717     }
718     cal->EdgeSlope[ramp][edge] = slope;
719     cal->EdgeOffset[ramp][edge] = (long)p[ramp][edge][0] -
720

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 7/154
LINE #		SOURCE TEXT		
721		(long)(cal->Edgeslope[ramp][edge]) * (long)t[ramp][edge][0],		
722		if(ramp == 0)		
723		nopooints = 3;		
724		else		
725		nopooints = 7;		
726		tmg_set_slot_count(nopooints + 1);		
727		p[ramp][edge][0] = tmg_predict_period(Measure_points[ramp].high,		
728		cal->Edgeslope[ramp][edge],		
729		cal->EdgeOffset[ramp][edge])/nopooints;		
730		try_t1 = tmg_predict_threshold(p[ramp][edge][0],		
731		cal->Edgeslope[ramp][edge],		
732		cal->EdgeOffset[ramp][edge]);		
733		count = 0;		
734		do {		
735		t[ramp][edge][0] = try_t1--;		
736		if (++count > 10) {		
737		tmg_report_failure(me,		
738		"Unusual ramp thresholds: tmg_find_period");		
739		return(FAILURE);		
740		if ((t[ramp][edge][0] = tmg_find_period(ramp, edge,		
741		(long)t[ramp][edge][0]))		
742		{		
743		tmg_report_failure(me,		
744		"can't find new low period: tmg_find_period");		
745		return(FAILURE);		
746		t[ramp][edge][nopooints-1] = tmg_predict_threshold(p[ramp][edge][0] * nopooints,		
747		cal->Edgeslope[ramp][edge],		
748		cal->EdgeOffset[ramp][edge]);		
749		} while (t[ramp][edge][nopooints-1] > 251);		
750		if ((t[ramp][edge][nopooints-1] = tmg_find_precise_period(ramp, edge,		
751		(long)t[ramp][edge][nopooints-1], (long)p[ramp][edge][0], nopooints)))		
752		{		
753		tmg_report_failure(me,		
754		"can't find high period: tmg_find_precise_period");		
755		tmg_set_slot_count(2);		
756		return(FAILURE);		
757		/* do rest of points */		
758		for(point = 1; point < nopooints - 1; point++)		
759		{		
760		t[ramp][edge][point] = tmg_predict_threshold(p[ramp][edge][0] * (point+1),		
761		cal->Edgeslope[ramp][edge],		
762		cal->EdgeOffset[ramp][edge]);		
763		if ((t[ramp][edge][point] = tmg_find_precise_period(ramp, edge,		
764		(long)t[ramp][edge][point], (long)p[ramp][edge][0], point + 1)))		
765		{		
766		tmg_report_failure(me,		
767		"can't find mid period: tmg_find_precise_period");		
768		tmg_set_slot_count(2);		
769		return(FAILURE);		
770		tmg_set_slot_count(2);		
771		if(ramp == 0)		
772		{		
773		plist = p[ramp][edge];		
774		tlist = t[ramp][edge];		
775		nopooints = nopooints;		
776		} else		
777		{		
778		plist = t(ramp)[edge][2];		
779		tlist = t(ramp)[edge][2];		
780		nopooints = nopooints - 2;		
781		} temperrors += (calculate_slope_and_offset_and_linearity(ramp,edge,cal,		
782		tlist,plist,nopooints,0);		
783		== SUCCESS) ? 0 : 1;		
784		*errors += temperrors;		
785		return (temperrors ? FAILURE : SUCCESS);		
786		}		
787		tmg_measure_offset(ramp, edge, cal)		
788		register long ramp, edge;		
789		register struct CALIS *cal;		
790		{		
791		static char me[] = "tmg_measure_offset";		
792		long period;		
793		long threshold;		
794		long offset;		
795		in_message("-");		
796		#ifdef DIAGS		
797		if (tmg_cal_debug & 2)		
798		in_message("tmg_measure_offset(td,td,tz)\n", ramp, edge, cal);		
799		#endif DIAGS		
800		/*		
801		.. 1. Find a point near the high end of the ramp.		
802		.. 2. Measure delay of that point.		
803		.. 3. Using known slope, calculate the offset of that line.		
804		*/		
805		tmg_set_slot_count(2);		
806		threshold = 250;		
807		if ((period = tmg_find_period(ramp, edge, threshold))) {		
808		tmg_report_failure(me,		
809		"can't find low period: tmg_find_period");		
810		return(FAILURE);		
811		cal->EdgeOffset[ramp][edge] = tmg_predict_offset(period, threshold,		
812		cal->Edgeslope[ramp][edge]);		
813		if (tmg_cal_debug & 8)		
814		in_message("ramp td edge td offset is td ps (td p td t td s)\n",		
815		ramp, edge, cal->EdgeOffset[ramp][edge],		
816		period, threshold, cal->Edgeslope[ramp][edge]);		
817		return SUCCESS;		
818		}		
819		calculate_slope_and_offset_and_linearity(ramp,edge,cal,t,p,nopooints,ramp)		
820		register long ramp, edge;		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE 5/23/89  
TIME 4:41:30 pm  
PAGE # 8/155

```

LINE # SOURCE TEXT
841 long t[]; p[], numpoints, armp;
842 struct CALIS "cal;
843 {
844     static char ms[] = "calculate_slope_and_offset_and_linearity";
845     void fit();
846     float midpredicted, error, foffset, fslope;
847     long period;
848     long slope; /* picosec/threshold */
849     long offset; /* picoseconds */
850     long linearity; /* millipercent */
851     register int i;
852     long slope_delta, ideal_slope;
853     int nonlinear;
854
855     float x[7], y[7], aiga, aigh, chl2, q, "xx, "yy;
856 #ifdef LOG_DEFINED
857     float sigy[7], "asigy;
858     long p, junk;
859     asigy = sigy + 1;
860 #endif LOG_DEFINED
861
862     xx = x - 1;
863     yy = y - 1;
864     for(i = 0; i < numpoints; i++)
865     {
866         x[i] = (float) t[i];
867         y[i] = (float) p[i];
868     }
869
870 #ifdef DIAGS
871     if (tmg_cal_debug & 0x1000000)
872     {
873         for(i = 0; i < numpoints; i++)
874             lm_message("x[%d] = %f ", i, x[i]);
875         lm_message("\n");
876         for(i = 0; i < numpoints; i++)
877             lm_message("y[%d] = %f ", i, y[i]);
878         lm_message("\n");
879     }
880 #endif DIAGS
881
882 #ifdef LOG_DEFINED
883     for(i = 0; i < numpoints; i++)
884     {
885         if (tmg_get_next_lower_period(p[i],
886             &p, &junk, &junk, &junk) != SUCCESS) {
887             tmg_report_failure(ms, "tmg_get_frequency_setting(1)");
888             return FAILURE;
889         }
890         sigy[i] = (p[i] - p);
891     }
892
893     fit(xx, yy, numpoints, asigy, 1, &foffset, &fslope, &aiga, &aigh, &chl2, &q);
894     linearity = (q * 1000.0);
895 #else LOG_DEFINED
896     fit(xx, yy, numpoints, 0, 0, &foffset, &fslope, &aiga, &aigh, &chl2, &q);
897 #endif LOG_DEFINED
898
899     slope = fslope;
900     offset = foffset;
901
902     midpredicted = foffset + (float)t[numpoints >> 1] * fslope;
903     miderror = fabs((midpredicted - (float)p[numpoints >> 1]) / (float)p[numpoints >> 1]);
904     linearity = (long)((100000 + miderror));
905
906     if (tmg_cal_debug & 0x1000000) {
907         lm_message("slope = %d offset = %d linearity = %d numpoints = %d\n",
908             slope, offset, linearity, numpoints);
909     }
910
911     switch(edge) {
912         case 0:
913             cal->EdgeOffset[ramp][edge] = offset;
914             cal->EdgeSlope[ramp][edge] = slope;
915             cal->EdgeLinearity[ramp][edge] = linearity;
916             ideal_slope = Measure_points[ramp].slope;
917             break;
918         case 5: /* really edge 7 */
919             cal->Edge7Slope[ramp] = slope;
920             cal->Edge7Linearity[ramp] = linearity;
921             ideal_slope = Measure_edge7_points[ramp].slope;
922             break;
923         case 7: /* really sample */
924             cal->SampleOffset[ramp][armp] = offset;
925             cal->SampleSlope[ramp] = slope;
926             cal->SampleLinearity[ramp] = linearity;
927             ideal_slope = Measure_sample_points[armp].slope;
928             break;
929         default:
930             lm_error("%s: Illegal edge %d\n", ms, edge);
931             break;
932     }
933
934     nonlinear = (linearity > 2500); /* 2500mV == 2.3% */
935     if (nonlinear) {
936         lm_error("Ramp %d Edge %d slope %d failed linearity (%dmt)\n",
937             ramp, edge, slope, linearity);
938     }
939
940 #ifdef DIAGS
941     if ((tmg_cal_debug & 0x2000000) || nonlinear) {
942         lm_message("\ncalculate_slope_and_offset_and_linearity(%d,%d):\n",
943             ramp, edge);
944         for(i = 0; i < numpoints; i++)
945         {
946             lm_message("t[%d] = %d p[%d] = %d ftd = %g fpid = %e\n",
947                 i, t[i], i, p[i], i, y[i]);
948         }
949         lm_message("fslope = %e slope = %d\n", fslope, slope);
950         lm_message("foffset = %e offset = %d\n", foffset, offset);
951         lm_message("midpredicted = %f linearity = %dmt\n",
952             midpredicted, linearity);
953         lm_message("miderror = %f\n", miderror);
954         lm_message("period = %d\n", period);
955     }
956     period = tmg_predict_period(t[numpoints >> 1], slope, offset);
957

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 9/156
LINE #	SOURCE TEXT			
961	in_message("threshold = %d\n",			
962	tmg_predict_threshold(period, slope, offset));			
963	}			
964	#endif DIAGS			
965	if (scaling == 1)			
966	return SUCCESS;			
967	}			
968	slope_delta = ideal_slope - 0.15; /* 13% */			
969	if ((slope < (ideal_slope - slope_delta))			
970	((slope > (ideal_slope + slope_delta))) {			
971	in_error("Ramp to edge to slope to out of range (%d +/- 15%)\n",			
972	(edge == 1)? ramp: ramp, edge, slope, ideal_slope);			
973	return FAILURE;			
974	}			
975	else			
976	return SUCCESS;			
977	}			
978	/*			
979	tmg_set_period(ramp, period);			
980	/* sets the clock to the closest period >= PERIOD psec and waits			
981	/* for the pll to settle			
982	*/			
983	#define CLOCK_SET_TIMEOUT 10000			
984	tmg_set_period(ramp, period)			
985	long ramp, *period;			
986	{			
987	long timeout;			
988	static char me[] = "tmg_set_period";			
989	long jitter, a, k, select;			
990	long actual;			
991	if (tmg_get_frequency_setting(*period,			
992	actual, &jitter, &a, &k, &select) != SUCCESS) {			
993	tmg_report_failure(me, "tmg_get_frequency_setting");			
994	return FAILURE;			
995	}			
996	#endif DIAGS			
997	if (tmg_cal_debug & 0x08) {			
998	in_message("%s: period to actual to psec a = %d k = %d sel = %d\n",			
999	me, "period, actual, a, k, select);			
1000	#endif DIAGS			
1001	period = actual;			
1002	if (!tmg_set_frequency((u_char)a, (u_char)k, (u_char)select) != SUCCESS)			
1003	{			
1004	tmg_report_failure(me, "tmg_set_frequency");			
1005	return FAILURE;			
1006	}			
1007	/* Set the sample pulse width so the pcc doesn't break */			
1008	/* Do this while the pll settles */			
1009	tmg_set_sample_width(actual, Measure_points[ramp].slope,			
1010	(long)TMC_MIN_SAMPLE_WIDTH);			
1011	for (timeout = CLOCK_SET_TIMEOUT;			
1012	timeout && (!tmg_check_locked()) != SUCCESS;			
1013	--timeout)			
1014	{			
1015	in_error("%s: CLOCK SET TIMED OUT\n", me);			
1016	return FAILURE;			
1017	}			
1018	return SUCCESS;			
1019	}			
1020	tmg_set_sample_width(period, slope, pw)			
1021	long period, slope, pw;			
1022	{			
1023	register long sample_start_time, sample_width;			
1024	if (pw < TMC_MIN_SAMPLE_WIDTH) return FAILURE;			
1025	sample_start_time = 2 * 512 * slope;			
1026	sample_width = (sample_start_time + pw)/period;			
1027	if (sample_width > 255)			
1028	sample_width = 255;			
1029	if (tmg_cal_debug & 0x10000000)			
1030	in_message("period = %d, sample width = %d (%d ps)\n",			
1031	period, sample_width, sample_width * period);			
1032	tmg_set_sample_width_reg(sample_width);			
1033	return SUCCESS;			
1034	}			
1035	#define MAX_THRESHOLD 50 /* 250 mv */			
1036	tmg_measure_minth(ramp, edge, threshold)			
1037	long ramp, edge, *threshold;			
1038	{			
1039	static char me[] = "tmg_measure_minth";			
1040	long period = 100175; /* 9.21 MHz == 100.375 ns */			
1041	long found_high;			
1042	if (tmg_clockoff() != SUCCESS) {			
1043	in_error("%s: find_min_th: could not turn clock off\n");			
1044	return FAILURE;			
1045	}			
1046	tmg_set_test_mode(1); tmg_set_slot_count(1);			
1047	for (*threshold = 4; *threshold <= MAX_THRESHOLD; ++*threshold) {			
1048	if (tmg_detect_always_high(ramp, edge, *period,			
1049	*threshold, 100001, &found_high) != SUCCESS) {			
1050	tmg_report_failure(me, "tmg_detect_always_high");			
1051	return FAILURE;			
1052	}			
1053	if (found_high) {			
1054	break;			
1055	}			
1056	if (*threshold > MAX_THRESHOLD) {			
1057	in_error("%s: testmode 1: no min threshold for edge to below %d\n",			
1058	me, edge, MAX_THRESHOLD);			
1059	return FAILURE;			
1060	}			
1061	if (tmg_cal_debug & 2)			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/trng_cal.c	DATE 5/23/89	PAGE # 10/157
LINE #		SOURCE TEXT		
1081		lm_message("Ramp td edge td min threshold = %d\n", ramp, edge, "threshold");		
1082		#endif DIAGS		
1083		if (trng_cal_debug & 1)		
1084		trng_play_til_key();		
1085		#endif DIAGS		
1086		/* Now do double ramp trng_set_test_mode(2) test. */		
1087		trng_set_test_mode(2); trng_set_slot_count(1);		
1088		for( threshold = MAX_THRESHOLD; --threshold; )		
1089		if (trng_detect_always_low(ramp, edge, &period,		
1090		threshold, 100001, &found_high) != SUCCESS) {		
1091		trng_report_failure(me, "trng_detect_always_low");		
1092		trng_set_test_mode(1); trng_set_slot_count(2);		
1093		return FAILURE;		
1094		}		
1095		if (found_high) {		
1096		break;		
1097		}		
1098		if (threshold > MAX_THRESHOLD) {		
1099		lm_error("%s: testmode 2: no min threshold for edge td below %d\n",		
1100		me, edge, MAX_THRESHOLD);		
1101		trng_set_test_mode(1); trng_set_slot_count(2);		
1102		return(FAILURE);		
1103		}		
1104		if(trng_cal_debug & 8)		
1105		lm_message("Ramp td edge td min threshold = %d after testmode 2\n", ramp, edge, "threshold");		
1106		#endif DIAGS		
1107		if (trng_cal_debug & 1)		
1108		trng_play_til_key();		
1109		#endif DIAGS		
1110		trng_set_test_mode(1); trng_set_slot_count(2);		
1111		return(SUCCESS);		
1112		}		
1113		trng_measure_edge7_sample_minth(aramp, aramp, thresh7, threshs)		
1114		long eramp, aramp, "thresh7, "threshs;		
1115		{		
1116		static char me[] = "trng_measure_edge7_sample_minth";		
1117		long period = 100000;		
1118		long found_high;		
1119		if(trng_clockoff() != SUCCESS) {		
1120		lm_error("%s:trng_measure_edge7_sample_minth: could not turn clock off\n");		
1121		return(FAILURE);		
1122		}		
1123		for(thresh7 = 4; thresh7 <= MAX_THRESHOLD; ++thresh7) {		
1124		lm_message("%s",		
1125		if(trng_detect_sample_always_high(aramp, aramp, &period, "thresh7, 101,		
1126		100001, &found_high) != SUCCESS) {		
1127		trng_report_failure(me, "trng_detect_sample_always_high");		
1128		goto cleanup;		
1129		}		
1130		if(found_high)		
1131		break;		
1132		#endif DIAGS		
1133		if (trng_cal_debug & 0x100)		
1134		trng_play_til_key();		
1135		#endif DIAGS		
1136		if (thresh7 > MAX_THRESHOLD) {		
1137		lm_error("%s: no min threshold for edge 7 below %d\n",		
1138		me, MAX_THRESHOLD);		
1139		goto cleanup;		
1140		}		
1141		/* lm_message("Ramp td edge 7 min threshold = %d\n", aramp, "thresh7"); */		
1142		for(threshs = 4; threshs <= MAX_THRESHOLD; ++threshs) {		
1143		lm_message("%s",		
1144		if(trng_detect_sample_always_high(aramp, aramp, &period, 101, "threshs,		
1145		100001, &found_high) != SUCCESS) {		
1146		trng_report_failure(me, "trng_detect_sample_always_high");		
1147		goto cleanup;		
1148		}		
1149		if(found_high)		
1150		break;		
1151		#endif DIAGS		
1152		if (trng_cal_debug & 0x100)		
1153		trng_play_til_key();		
1154		#endif DIAGS		
1155		if (threshs > MAX_THRESHOLD) {		
1156		lm_error("%s: no min threshold for Sample below %d\n",		
1157		me, MAX_THRESHOLD);		
1158		goto cleanup;		
1159		}		
1160		/* lm_message("Ramp td edge 7 min threshold = %d\n", 0, "threshs"); */		
1161		return SUCCESS;		
1162		}		
1163		cleanup:		
1164		return FAILURE;		
1165		}		
1166		trng_detect_sample(eramp, aramp, mode, period, thresh7, threshs, count, actual)		
1167		register long eramp, aramp;		
1168		long *period, *actual, thresh7, threshs, count;		
1169		int mode;		
1170		{		
1171		static char me[] = "trng_detect_sample";		
1172		trng_set_eramp(eramp);		
1173		trng_set_aramp(aramp);		
1174		trng_set_edge_7_threshold(thresh7);		
1175		trng_set_sample_threshold(threshs);		
1176		if (trng_set_period(eramp, period) != SUCCESS) {		
1177		}		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE	5/23/89	PAGE #
TIME	4:41:30 pm	11/158

```

1201 tmg_report_failure(me, "tmg_set_period");
1202 return FAILURE;
1203 }
1204
1205 #ifdef DIAGS
1206     if (tmg_cal_debug & 0x100) {
1207         lm_message("tmg_detect_sample: td psec\n", "period");
1208     }
1209 #endif DIAGS
1210
1211     if (tmg_detect(0l, count, actual, mode) != SUCCESS) {
1212         tmg_report_failure(me, "tmg_detect");
1213         return FAILURE;
1214     }
1215 #ifdef DIAGS
1216     if (tmg_cal_debug & 0x100)
1217         lm_message("tmg_detect_sample: count = %d\n", "actual");
1218 #endif DIAGS
1219     return SUCCESS;
1220 }
1221
1222 /*
1223  ** tmg_detect_always_low() is a magic function that reads the edge register
1224  ** up to "count" times and returns 1 if the value is always 0.
1225  ** Is our application remember that a sometimes high
1226  ** is as good as always high, so this function returns 0 upon finding
1227  ** the first high.
1228  **
1229  ** In searching for the threshold period, we will search for the shortest
1230  ** period that sometimes sees high.
1231  **
1232  ** ----- this is the edge we will seek.
1233  */
1234
1235 tmg_detect_edge(ramp, edge, mode, period, threshold, count, actual)
1236 register long ramp, edge;
1237 register long *period, *actual;
1238 long threshold, count;
1239 {
1240     static char me[] = "tmg_detect_edge";
1241
1242     tmg_set_xramp(ramp);
1243     tmg_set_threshold(edge, threshold);
1244
1245     if (tmg_set_period(ramp, period) != SUCCESS) {
1246         tmg_report_failure(me, "tmg_set_period");
1247         return FAILURE;
1248     }
1249 #ifdef DIAGS
1250     if (tmg_cal_debug & 2) {
1251         lm_message("tmg_detect_edge: td psec\n", "period");
1252     }
1253 #endif DIAGS
1254
1255     if (tmg_detect(edge, count, actual, mode) != SUCCESS) {
1256         tmg_report_failure(me, "tmg_detect");
1257         return FAILURE;
1258     }
1259 #ifdef DIAGS
1260     if (tmg_cal_debug & 8)
1261         lm_message("tmg_detect_edge: count = %d\n", "actual");
1262 #endif DIAGS
1263     return SUCCESS;
1264 }
1265
1266 int tmg_detect(edge, repcount, count, mode)
1267 register long edge, repcount, *count;
1268 int mode;
1269 {
1270     register long mask;
1271     static char me[] = "tmg_detect";
1272     register long previous = 0, this;
1273
1274     *count = 0;
1275     if (tmg_effclear() != SUCCESS) {
1276         lm_error("tmg_effclear: cal flip flops did not clear\n");
1277         tmg_report_failure(me, "tmg_effclear");
1278         return FAILURE;
1279     }
1280
1281     mask = 1 << edge;
1282
1283     if (tmg_cal_debug & 2)
1284         printf("tmg_detect(%d,%d,%d,%d)\n", edge, repcount, count, mode);
1285     while (repcount-- > 0) {
1286         if (tmg_play((long) TIMEOUT) != SUCCESS) {
1287             tmg_report_failure(me, "tmg_play");
1288             return FAILURE;
1289         }
1290         if (mode & DETECT_SAMPLE)
1291             this = (tmg_get_sample_cal());
1292         else
1293             this = (tmg_get_edge_cal() & mask);
1294         if ((this == previous) != ((mode & DETECT_LOW) != 0)) {
1295             if (mode & DETECT_BOOL) {
1296                 *count = 0;
1297             }
1298             #ifdef DIAGS
1299             if (tmg_cal_debug & 4)
1300                 lm_message("td: repcount = %d\n",
1301                     me, repcount);
1302             #endif DIAGS
1303             return SUCCESS; /* Found a wrong */
1304         } else {
1305             ++*count;
1306         }
1307         previous = this;
1308     }
1309     if (mode & DETECT_BOOL) {
1310         *count = (*count != 0);
1311     }
1312     return SUCCESS; /* Found all rights */
1313 }
1314
1315 }
1316
1317 }
1318
1319 }
1320

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE 5/23/89

PAGE #

TIME 4:41:30 pm

12/159

```

LINE # SOURCE TEXT
1321 #define SEARCH_COUNT 100
1322
1323 tmg_find_period(ramp, edge, thr)
1324 long ramp, edge, thr;
1325 {
1326     static char me[] = "tmg_find_period";
1327     long period, count;
1328     register long upper_bound, lower_bound;
1329
1330 #ifdef info
1331     1. Select the first period such that for the fastest possible
1332        ramp or the slowest possible ramp, the ramp discharges
1333        during the inactive phase of the first cycle (Set period
1334        to 1/3 longer than expected ramp speed);
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356 #endif info
1357     upper_bound = (4 * Measure_points[ramp].slope * 255)/3;
1358     /* If (upper_bound < MIN_PERIOD) then something is wrong */
1359
1360 #ifdef DIAGS
1361     if (tmg_cal_debug & 4)
1362         lm_message("tmg_find_period: starting at %d psec\n",
1363             upper_bound);
1364 #endif DIAGS
1365
1366     /* correct for first time through loop: */
1367     period = (100 * upper_bound)/tmg_reduction_factor;
1368     do {
1369         upper_bound = period;
1370         period = (tmg_reduction_factor * period)/100;
1371         if (period < MIN_PERIOD)
1372             period = MIN_PERIOD;
1373         if (tmg_detect_always_low(ramp, edge,
1374             &period, thr, (long)SEARCH_COUNT, &count) != SUCCESS) {
1375             tmg_report_failure(me, "tmg_detect_always_low(1)");
1376             return 0;
1377         }
1378 #ifdef DIAGS
1379         if (tmg_cal_debug & 1)
1380             tmg_play_til_key();
1381 #endif DIAGS
1382         if (period == upper_bound) {
1383             lm_error("tmg_find_period: Could not find low\n");
1384             return 0;
1385         }
1386     } while (!count);
1387 #ifdef DIAGS
1388     if (tmg_cal_debug & 4)
1389         lm_message("found low at %d\n", period);
1390 #endif DIAGS
1391
1392     /* one more time... */
1393     do {
1394         upper_bound = period;
1395         period = (tmg_reduction_factor * period)/100;
1396         if (period < MIN_PERIOD)
1397             period = MIN_PERIOD;
1398         if (tmg_detect_always_low(ramp, edge,
1399             &period, thr, (long)SEARCH_COUNT, &count) != SUCCESS) {
1400             tmg_report_failure(me, "tmg_detect_always_low(2)");
1401             return 0;
1402         }
1403 #ifdef DIAGS
1404         if (tmg_cal_debug & 1)
1405             tmg_play_til_key();
1406 #endif DIAGS
1407         if (period == upper_bound) {
1408             lm_error("tmg_find_period: Could not find 2nd high\n");
1409             return 0;
1410         }
1411     } while (!count);
1412 #ifdef DIAGS
1413     if (tmg_cal_debug & 4)
1414         lm_message("found second high at %d\n", period);
1415 #endif DIAGS
1416     lower_bound = period;
1417     return tmg_edge_search(ramp, edge, thr, upper_bound, lower_bound);
1418 }
1419
1420 tmg_edge_search(ramp, edge, thr, upper_bound, lower_bound)
1421 long ramp, edge, thr, upper_bound, lower_bound;
1422 {
1423     static char me[] = "tmg_edge_search";
1424     long period, found_high;
1425     long timeout = BIN_LIMIT;
1426
1427     for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {
1428         period = (upper_bound + lower_bound)/2;
1429         if (tmg_detect_always_low(ramp, edge, &period,
1430             thr, (long)SEARCH_COUNT, &found_high) != SUCCESS) {
1431             tmg_report_failure(me, "tmg_detect_always_low(3)");
1432         }
1433     }
1434 #ifdef DIAGS
1435     if (tmg_cal_debug & 1)
1436         tmg_play_til_key();
1437 #endif DIAGS
1438     if ((period <= lower_bound) || (period >= upper_bound)) {
1439         /* We can't get more exact than this */
1440 #ifdef DIAGS

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 13/160
LINE #	SOURCE TEXT			
1441	if (tmg_cal_debug & 4)			
1442	lm_message("(got it between td and td)\n",			
1443	lower_bound, upper_bound);			
1444	#endif DIAGS			
1445	return period;			
1446	}			
1447	if (!found_high) {			
1448	/* too short - raise lower bound */			
1449	lower_bound = period;			
1450	} else {			
1451	/* too long - lower upper bound */			
1452	upper_bound = period;			
1453	}			
1454	}			
1455	lm_error("BINARY EDGE SEARCH FAILED TO CONVERGE!\n");			
1456	return period;			
1457	}			
1458	}			
1459	tmg_find_precise_period(ramp, edge, thr, base_period, multiplier)			
1460	long ramp, edge, thr, base_period, multiplier;			
1461	{			
1462	static char me[] = "tmg_find_precise_period";			
1463	long period, count1, count2;			
1464	register long upper_bound, lower_bound, delta;			
1465	#ifdef info			
1466	1. Select the first period based on the predicted base value.			
1467	2. Compute delta based on the period and multiplier.			
1468	3. If you detect low, decrease period by delta. If you still			
1469	detect low, then error out.			
1470	4. If you detect high, increase period by delta. If you still			
1471	detect high, then error out.			
1472	5. Do a binary search for the exact low-to-high period.			
1473	#endif info			
1474	period = base_period;			
1475	#ifdef DIAGS			
1476	if (tmg_cal_debug & 0x2)			
1477	lm_message("tmg_find_precise_period(td,td,td,td)\n",			
1478	ramp, edge, thr, base_period, multiplier);			
1479	#endif			
1480	if (tmg_cal_debug & 4)			
1481	lm_message("ts: starting at td psec\n", me, period);			
1482	#endif DIAGS			
1483	delta = (period / 4) / multiplier;			
1484	{			
1485	if (tmg_detect_always_low(ramp, edge,			
1486	period, thr, (long)SEARCH_COUNT, count1) != SUCCESS) {			
1487	tmg_report_failure(me, "tmg_detect_always_low(1)");			
1488	return 0;			
1489	#endif DIAGS			
1490	if (tmg_cal_debug & 1)			
1491	tmg_play_til_key();			
1492	#endif DIAGS			
1493	if (!count1) {			
1494	lower_bound = period;			
1495	period += delta;			
1496	} else {			
1497	upper_bound = period;			
1498	period -= delta;			
1499	}			
1500	}			
1501	if (tmg_detect_always_low(ramp, edge,			
1502	period, thr, (long)SEARCH_COUNT, count2) != SUCCESS) {			
1503	tmg_report_failure(me, "tmg_detect_always_low(2)");			
1504	return 0;			
1505	#endif DIAGS			
1506	if (tmg_cal_debug & 1)			
1507	tmg_play_til_key();			
1508	#endif DIAGS			
1509	if (!count2) {			
1510	lower_bound = period;			
1511	} else {			
1512	upper_bound = period;			
1513	}			
1514	}			
1515	if (count1 == count2) {			
1516	lm_error("ts: ts on ramp td edge td\n", me,			
1517	"Can't find edge boundaries", ramp, edge);			
1518	return 0;			
1519	}			
1520	return tmg_edge_search(ramp, edge, thr, upper_bound, lower_bound)			
1521	* multiplier;			
1522	}			
1523	}			
1524	tmg_sample_search(aramp, aramp, thr7, thr, upper_bound, lower_bound)			
1525	long aramp, aramp, thr7, thr, upper_bound, lower_bound;			
1526	{			
1527	static char me[] = "tmg_sample_search";			
1528	long period, found_high;			
1529	long timeout = BIN_LIMIT;			
1530	for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {			
1531	period = (upper_bound + lower_bound)/2;			
1532	if (tmg_detect_sample_always_low(aramp, aramp, (long*)period,			
1533	thr7, thr, (long)SEARCH_COUNT, (long*)found_high)			
1534	!= SUCCESS)			
1535	{			
1536	tmg_report_failure(me,			
1537	"tmg_detect_sample_always_low(3)");			
1538	}			
1539	#ifdef DIAGS			
1540	if (tmg_cal_debug & 0x100)			
1541	tmg_play_til_key();			
1542	#endif DIAGS			
1543	if ((period <= lower_bound)    (period >= upper_bound)) {			
1544	/* We can't get more exact than this */			
1545	#endif DIAGS			
1546	if (tmg_cal_debug & 0x400)			
1547	lm_message("(got it between td and td)\n",			
1548	lower_bound, upper_bound);			
1549	#endif DIAGS			
1550	return period;			
1551	}			
1552	if (!found_high) {			
1553	/* too short - raise lower bound */			
1554	}			
1555	}			
1556	}			
1557	}			
1558	}			
1559	}			
1560	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

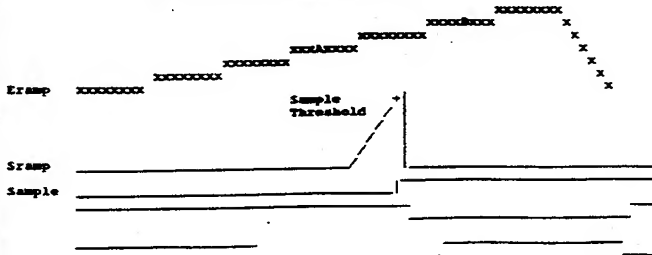
DATE 5/23/89  
TIME 4:41:30 pm

PAGE #  
14/161

```

LINE #          SOURCE TEXT
1561         lower_bound = period;
1562     } else {
1563         /* too long - lower upper bound */
1564         upper_bound = period;
1565     }
1566 }
1567 lm_error("PRIMARY SAMPLE SEARCH FAILED TO CONVERGE!\n");
1568 return period;
1569 }
1570 /* tmg_sample_search */
1571 tmg_find_precise_sample_period(arm, armp, thr7, thrs, base_period, multiplier)
1572 long arm, armp, thr7, thrs, base_period, multiplier;
1573 {
1574     static char me[] = "tmg_find_precise_sample_period";
1575     long period, count1, count2;
1576     register long upper_bound, lower_bound, delta;
1577
1578 #ifdef INFO
1579     1. Select the first period based on the predicted base value.
1580     2. Compute delta based on the period and multiplier.
1581     3. If you detect low, decrease period by delta. If you still
1582        detect low, then error out.
1583     4. If you detect high, increase period by delta. If you still
1584        detect high, then error out.
1585     5. Do a binary search for the exact low-to-high period.
1586 #endif INFO
1587 period = base_period;
1588 #ifdef DIAGS
1589     if (tmg_cal_debug & 0x200)
1590         lm_message("tmg_find_precise_sample_period(%d,%d,%d,%d,%d)\n",
1591             arm, armp, thr7, thrs, base_period, multiplier);
1592     if (tmg_cal_debug & 0x1000)
1593         lm_message("ts: starting at %d psec\n", me, period);
1594 #endif DIAGS
1595     delta = (period / 4) / multiplier;
1596     {
1597         if (tmg_detect_sample_always_low(arm, armp,
1598             period, thr7, thrs, (long)SEARCH_COUNT, &count1) != SUCCESS) {
1599             tmg_report_failure(me,
1600                 "tmg_detect_sample_always_low(1)");
1601             return 0;
1602         }
1603     }
1604 #ifdef DIAGS
1605     if (tmg_cal_debug & 0x100)
1606         tmg_play_til_key();
1607 #endif DIAGS
1608     if (!count1) {
1609         lower_bound = period;
1610         period -= delta;
1611     } else {
1612         upper_bound = period;
1613         period += delta;
1614     }
1615     {
1616         if (tmg_detect_sample_always_low(arm, armp,
1617             period, thr7, thrs, (long)SEARCH_COUNT, &count2) != SUCCESS) {
1618             tmg_report_failure(me,
1619                 "tmg_detect_sample_always_low(2)");
1620             return 0;
1621         }
1622     }
1623 #ifdef DIAGS
1624     if (tmg_cal_debug & 0x100)
1625         tmg_play_til_key();
1626 #endif DIAGS
1627     if (!count2) {
1628         lower_bound = period;
1629     } else {
1630         upper_bound = period;
1631     }
1632     {
1633         if (count1 == count2) {
1634             lm_error("ats: ts on edge ramp to sample ramp to\n", me,
1635                 "Can't find sample boundaries", arm, armp);
1636             return 0;
1637         }
1638         return tmg_sample_search(arm, armp, thr7, thrs, upper_bound, lower_bound,
1639             multiplier);
1640     }
1641 }
1642 /* tmg_find_precise_sample_period */
1643 tmg_find_sample_period(arm, armp, thresh7, threshs)
1644 long arm, armp, thresh7, threshs;
1645 {
1646     static char me[] = "tmg_find_sample_period";
1647     long found_high, period, count;
1648     register long upper_bound, lower_bound;
1649     long timeout;
1650
1651 #ifdef INFO
1652     1. Select the first period such that for the fastest possible
1653        ramp or the slowest possible ramp, the ramp discharges
1654        during the inactive phase of the first cycle (set period
1655        to 1/3 longer than expected ramp speed);
1656 #endif INFO
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89 TIME 4:41:30 pm	PAGE # 15/162
LINE #	SOURCE TEXT			
1681	2. If you detect low, go to step 3 (see point B in the figure).			
1682	Otherwise (see point A in the figure), keep reducing			
1683	the period by .5 until you detect a low.			
1684	3. Reduce the frequency by .5 until you get a high. Remember the last			
1685	frequencies that you got high and low.			
1686	4. Do a binary search for the low-to-high period.			
1687	Sendif info			
1688	if(threshs == 10) /* must be looking at edge? ramp */			
1689	upper_bound = (4 * Measure_edge_points[aramp].slope * 255)/3;			
1690	else			
1691	upper_bound = (4 * Measure_sample_points[aramp].slope * 255)/3 + 40000;			
1692	/* If (upper_bound < MIN_PERIOD) then something is wrong */			
1693	#endif DIAGS			
1694	if (tmg_cal_debug & 0x1000)			
1695	lm_message("tmg_find_sample_period: starting at %d msec\n",			
1696	upper_bound);			
1697	#endif DIAGS			
1698	/* correct for first time through loop: */			
1699	period = (100 * upper_bound)/tmg_reduction_factor;			
1700	do {			
1701	upper_bound = period;			
1702	period = (tmg_reduction_factor * period)/100;			
1703	if (period < MIN_PERIOD)			
1704	period = MIN_PERIOD;			
1705	if (tmg_detect_sample_always_low(aramp, aramp,			
1706	aperiod, threshs, (long)SEARCH_COUNT, &count) != SUCCESS) {			
1707	tmg_report_failure(me, "tmg_detect_sample_always_low(1)");			
1708	return 0;			
1709	#endif DIAGS			
1710	if (tmg_cal_debug & 0x100)			
1711	tmg_play_till_key();			
1712	#endif DIAGS			
1713	if (period == upper_bound) {			
1714	lm_error("tmg_find_sample_period: Could not find low\n");			
1715	return 0;			
1716	} while (!count);			
1717	#endif DIAGS			
1718	if (tmg_cal_debug & 0x100)			
1719	lm_message("found low at %d\n", period);			
1720	#endif DIAGS			
1721	do {			
1722	upper_bound = period;			
1723	period = (tmg_reduction_factor * period)/100;			
1724	if (period < MIN_PERIOD)			
1725	period = MIN_PERIOD;			
1726	if (tmg_detect_sample_always_low(aramp, aramp,			
1727	aperiod, threshs, (long)SEARCH_COUNT, &count) != SUCCESS) {			
1728	tmg_report_failure(me, "tmg_detect_sample_always_low(2)");			
1729	return 0;			
1730	#endif DIAGS			
1731	if (tmg_cal_debug & 0x100)			
1732	tmg_play_till_key();			
1733	#endif DIAGS			
1734	if (period == upper_bound) {			
1735	lm_error("tmg_find_sample_period: Could not find 2nd high\n");			
1736	return 0;			
1737	} while (!count);			
1738	#endif DIAGS			
1739	if (tmg_cal_debug & 0x100)			
1740	lm_message("found second high at %d\n", period);			
1741	#endif DIAGS			
1742	lower_bound = period;			
1743	for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {			
1744	period = (upper_bound + lower_bound)/2;			
1745	if (tmg_detect_sample_always_low(aramp, aramp, aperiod,			
1746	threshs, (long)SEARCH_COUNT, &found_high) != SUCCESS) {			
1747	tmg_report_failure(me, "tmg_detect_sample_always_low(3)");			
1748	#endif DIAGS			
1749	if (tmg_cal_debug & 0x100)			
1750	tmg_play_till_key();			
1751	#endif DIAGS			
1752	if ((period <= lower_bound)    (period >= upper_bound)) {			
1753	/* We can't get more exact than this */			
1754	#endif DIAGS			
1755	if (tmg_cal_debug & 0x100)			
1756	lm_message("got it between %d and %d\n",			
1757	lower_bound, upper_bound);			
1758	#endif DIAGS			
1759	return period;			
1760	} if (!found_high) {			
1761	/* too short - raise lower bound */			
1762	lower_bound = period;			
1763	} else {			
1764	/* too long - lower upper bound */			
1765	upper_bound = period;			
1766	}			
1767	lm_error("BINARY SAMPLE EDGE SEARCH FAILED TO CONVERGE!\n");			
1768	return period;			
1769	#endif			
1770	tmg_compute_delay(ramp, edge, cal, delay)			
1771	register long ramp, edge;			
1772	register long *delay;			
1773	register struct CALIS *cal;			
1774	{			
1775	static char me[] = "tmg_compute_delay";			
1776	long adjust;			
1777	/* We assume that this ramp has already been measured */			
1778	#endif			
1779	if (tmg_measure_offset(ramp, edge, cal) != SUCCESS) {			
1780	tmg_report_failure(me, "tmg_measure_offset");			
1781	return FAILURE;			
1782	}			
1783	adjust = MAGIC_DEAD_TIME			
1784	- cal->EdgeOffset[ramp][edge];			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 16/163
LINE #	SOURCE TEXT			
1801	- (long)(cal->Edgeslope[ramp][edge] - cal->EdgesinThreshold[ramp][edge]);			
1802				
1803	#ifdef DIAGS			
1804	if (tmg_cal_debug & 0x30000) {			
1805	lm_message("Give an offset of td psec, slope of td psec/thresh.\n",			
1806	cal->EdgesOk, ramp[edge], cal->Edgeslope[ramp][edge]);			
1807	lm_message("a threshold of td and an estimated dead time of td psec.\n",			
1808	cal->EdgesinThreshold[ramp][edge], MAGIC_DEAD_TIME);			
1809	lm_message("I think we should adjust the delay by td psec.\n",			
1810	adjust);			
1811	}			
1812	#endif DIAGS			
1813	delay = MAGIC_DEAD_TIME - cal->EdgesOffset[ramp][edge]			
1814	- cal->Edgeslope[ramp][edge] * cal->EdgesinThreshold[ramp][edge];			
1815	return(SUCCESS);			
1816	}			
1817				
1818				
1819	void			
1820	tmg_adjust_delay(delta)			
1821	register long delta; /* microseconds */			
1822	{			
1823	/* delay and delta are expressed in ns */			
1824	long delay = tmg_get_delay();			
1825	/*			
1826	#ifdef DIAGS			
1827	if (tmg_cal_debug & 0x00000)			
1828	lm_message("Present delay = td ns (td)\n", delay,			
1829	tmg_get_delay_reg());			
1830	#endif DIAGS			
1831	delay += delta/1000; /* microseconds */			
1832	#endif DIAGS			
1833	if (tmg_cal_debug & 0x00000)			
1834	lm_message("Setting delay to td ns\n", delay);			
1835	#endif DIAGS			
1836	/* should we do bounds checking here, bill? */			
1837	tmg_set_delay(delay);			
1838	/*			
1839	#ifdef DIAGS			
1840	if (tmg_cal_debug & 0x00000)			
1841	lm_message("New delay = td ns (td)\n", tmg_get_delay(),			
1842	tmg_get_delay_reg());			
1843	#endif DIAGS			
1844	}			
1845	/*			
1846	tmg_get_frequency_setting()			
1847	/*			
1848	This routine computes the values for n, k, and clock			
1849	select register setting to give the best approximation			
1850	of the desired clock period. The clock period provided			
1851	is always equal to or greater than that requested.			
1852	The actual clock period is related to n, k as follows:			
1853	period = (k / n) * T_REF, where			
1854	k = 1, 2, ..., 256 or k = 3, 4, ..., 512,			
1855	n = 128, 129, ..., 256, and			
1856	T_REF = 256 / 10 MHz PLL reference period.			
1857	Identical to tmg_get_frequency_setting without max bounds checking.			
1858	Returns: SUCCESS except if period violates min bound.			
1859	*/			
1860	#define I_T_REF 553333 /* reference period in ps */			
1861	#define K_CRITICAL 251			
1862	{			
1863	register long period; /* physical clock period in ps */			
1864	register long *actualptr; /* actual clock period in ps */			
1865	register long *jitterptr; /* jitter per. period in ps */			
1866	register long *nptr;			
1867	register long *kptr;			
1868	register long *selectptr;			
1869	{			
1870	register long estimate, error, besterror;			
1871	register int k, n, save_k = 0, save_n = 0;			
1872	register long pef = period/0x30000, sf;			
1873	if ((period < 33333)    (period > 3500000))			
1874	return(FAILURE);			
1875	for(besterror = 0x7fffffff, n = N_MIN, n <= N_MAX, n++) {			
1876	for ((error = (n * pef)/512), sf = 0, error, error >= 1) {			
1877	++sf;			
1878	/* use the best-approximate formula for this period. */			
1879	k = (n * (period >> sf)) / (I_T_REF >> sf);			
1880	if (k > K_CRITICAL)			
1881	estimate = k * (I_T_REF/n);			
1882	else			
1883	estimate = (k * I_T_REF)/n;			
1884	if (estimate < period)			
1885	++k;			
1886	if (k > 512) /* see if out of range */			
1887	continue; /* if so, ignore it */			
1888	if (k > 256) /* see if above 256 */			
1889	if (k & 1) /* see if odd */			
1890	k++; /* make even */			
1891	if (k > K_CRITICAL)			
1892	estimate = k * (I_T_REF/n);			
1893	else			
1894	estimate = (k * I_T_REF)/n;			
1895	if ((error = estimate - period) < 0)			
1896	if (error < besterror) {			
1897	save_n = n;			
1898	save_k = k;			
1899	besterror = error;			
1900	}			
1901	}			
1902	if (besterror == 0x7fffffff) {			
1903	return(FAILURE);			
1904	}			
1905	*nptr = 256 - save_n + 1; /* value to put in reg */			
1906				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE 5/23/89  
TIME 4:41:30 pm

PAGE #  
17/164

```

LINE # SOURCE TEXT
1921 if(save_k == 1) {
1922     *kptr = 255; /* special case */
1923     *selectptr = 0; /* select div by 2 */
1924 } else {
1925     if(save_k > 256) {
1926         *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
1927         *selectptr = 2; /* select div by 4k */
1928     } else {
1929         *kptr = 256 - save_k + 1; /* normal case */
1930         *selectptr = 1; /* select div by 2k */
1931     }
1932 }
1933 /* actualptr = T_REF * save_k / save_n */
1934 if (save_k > K_CRITICAL) {
1935     *actualptr = save_k * (I_T_REF/save_n);
1936 } else {
1937     *actualptr = (save_k * I_T_REF)/save_n;
1938 }
1939 *jitterptr = 20 * 2 * save_k; /* 20 ps per PLL clock */
1940 return(SUCCESS);
1941 }
1942
1943 tmg_get_best_lower_period(period,actualptr,jitterptr,sptr,kptr,selectptr)
1944 register long period; /* physical clock period in ps */
1945 register long *actualptr; /* actual clock period in ps */
1946 register long *jitterptr; /* jitter per period in ps */
1947 register long *sptr;
1948 register long *kptr;
1949 register long *selectptr;
1950 {
1951     register long estimate, error, besterror;
1952     register int k, n, save_k = 0, save_n = 0;
1953     register long paf = period/0x10000, af;
1954
1955     if ((period < 33333) || (period > 35000000))
1956         return(FAILURE);
1957
1958     for(besterror = 0x7fffffff, n = N_MIN, k = N_MAX, n++) {
1959         for ((error = (n * paf)/512), af = 0, error; error >= 1) {
1960             ++af;
1961             /* use the most accurate formula for this period... */
1962             k = (n * (period >> af)) / (I_T_REF >> af);
1963             if (k > K_CRITICAL)
1964                 estimate = k * (I_T_REF/n);
1965             else
1966                 estimate = (k * I_T_REF)/n;
1967             if (estimate >= period)
1968                 ++k;
1969             if (k > 512) /* see if out of range */
1970                 continue; /* if so, ignore it */
1971             if (k > 256) /* see if above 256 */
1972                 if(k & 1) /* see if odd */
1973                     k++; /* make even */
1974             if (k > K_CRITICAL)
1975                 estimate = k * (I_T_REF/n);
1976             else
1977                 estimate = (k * I_T_REF)/n;
1978             if ((error = period - estimate) <= 0)
1979                 continue;
1980             if (error < besterror) {
1981                 save_n = n;
1982                 save_k = k;
1983                 besterror = error;
1984             }
1985         }
1986     }
1987     if (besterror == 0x7fffffff) {
1988         return(FAILURE);
1989     }
1990     *kptr = 256 - save_n + 1; /* value to put in reg */
1991     if(save_k == 1) {
1992         *kptr = 255; /* special case */
1993         *selectptr = 0; /* select div by 2 */
1994     } else {
1995         if(save_k > 256) {
1996             *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
1997             *selectptr = 2; /* select div by 4k */
1998         } else {
1999             *kptr = 256 - save_k + 1; /* normal case */
2000             *selectptr = 1; /* select div by 2k */
2001         }
2002     }
2003     /* actualptr = T_REF * save_k / save_n */
2004     if (save_k > K_CRITICAL) {
2005         *actualptr = save_k * (I_T_REF/save_n);
2006     } else {
2007         *actualptr = (save_k * I_T_REF)/save_n;
2008     }
2009     *jitterptr = 20 * 2 * save_k; /* 20 ps per PLL clock */
2010     return(SUCCESS);
2011 }
2012
2013 /* threshold and period array for sample calibration */
2014 static long st[NUMBER_OF_ERASES][NUMBER_OF_SAMPLES][7];
2015 static long sp[NUMBER_OF_ERASES][NUMBER_OF_SAMPLES][7];
2016
2017 tmg_measure_sample_ramp(erase, ramp, cal, errors)
2018 register long erase, ramp, *errors;
2019 register struct CALLS *cal;
2020 {
2021     static char me[] = "tmg_measure_sample_ramp";
2022     char buffer[64];
2023     long t17, t27;
2024     register long s0points, slope, dp, dt, count, point, temperrors;
2025     long thresh, offset;
2026
2027 #ifdef DIAGS
2028     if (tmg_cal_debug & 0x200)
2029         lm_message("ts(td,tx)\n", me, ramp, cal);
2030 #endif
2031     /*
2032     ** 1. find a point above but near the minimum threshold and longer
2033     ** than the magic chip dead time.
2034     ** 2. find a point high on the ramp.
2035     ** 3. Calculate the slope and offset of that line.
2036     ** 3a. Use slope and offset to predict points for fine calibration.
2037     ** 3b. Results of fine calibration are used for slope and offset.
2038     ** 4. Find a point approx midway between the first two points
2039     ** using results of 3b.
2040     ** 5. Compute a linearity value.
2041     */

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 18/165
LINE #		SOURCE TEXT		
2041	2042	/* Slope will be stored as a long. The units are v/s (or uv/us).		
2043	2044	*/		
2045	2046	temperrors = 0;		
2047	2048	{		
2049	2050	case 0:		
2051	2052	sopoints = 3;		
2053	2054	break;		
2055	2056	case 2:		
2057	2058	sopoints = 7;		
2059	2060	break;		
2061	2062	tmg_set_slot_count(2);		
2063	2064	t17 = Measure_edge7_points[aramp].low;		
2065	2066	st[aramp][aramp][0] = Measure_sample_points[aramp].low;		
2067	2068	if (! (sp[aramp][aramp][0] = tmg_find_sample_period(aramp, aramp, t17,		
2069	2070	(long)st[aramp][aramp][0]))		
2071	2072	{		
2073	2074	tmg_report_failure(me,		
2075	2076	"Can't find low period: tmg_find_sample_period",		
2077	2078	return(FAILURE);		
2079	2080	}		
2081	2082	t17 = Measure_edge7_points[aramp].high;		
2083	2084	st[aramp][aramp][2] = Measure_sample_points[aramp].high;		
2085	2086	if (! (sp[aramp][aramp][2] = tmg_find_sample_period(aramp, aramp, t17,		
2087	2088	(long)st[aramp][aramp][2]))		
2089	2090	{		
2091	2092	tmg_report_failure(me,		
2093	2094	"Can't find high period: tmg_find_sample_period",		
2095	2096	return(FAILURE);		
2097	2098	}		
2099	2100	if ((dt = (st[aramp][aramp][2] - st[aramp][aramp][0]) == 0) {		
2101	2102	in_error("Infinite sample ramp slope\n");		
2103	2104	return FAILURE;		
2105	2106	}		
2107	2108	if ((dp = (sp[aramp][aramp][2] - sp[aramp][aramp][0]) == 0) {		
2109	2110	in_error("Zero sample ramp slope\n");		
2111	2112	return FAILURE;		
2113	2114	}		
2115	2116	/* slopes are normally 500, 1000, 2000, and 4000. */		
2117	2118	slope = dp/dt;		
2119	2120	if ((slope < 50)    (slope > 40000)) {		
2121	2122	in_error("Illegal sample ramp slope of %d pa/thr\n", slope);		
2123	2124	return FAILURE;		
2125	2126	}		
2127	2128	offset = sp[aramp][aramp][0] - slope*st[aramp][aramp][0];		
2129	2130	sp[aramp][aramp][0] = tmg_predict_period(Measure_sample_points[aramp].high,		
2131	2132	slope, offset) * 2 / (sopoints + 1);		
2133	2134	thresh = tmg_predict_threshold((long)sp[aramp][aramp][0],		
2135	2136	slope, offset);		
2137	2138	st[aramp][aramp][0] = thresh;		
2139	2140	if (st[aramp][aramp][0] < cal->SampleMinThresh(aramp))		
2141	2142	{		
2143	2144	errors++;		
2145	2146	sprintf(buffer, "Can't fit %d periods in sample ramp of", sopoints, aramp);		
2147	2148	tmg_report_failure(me, buffer);		
2149	2150	return(FAILURE);		
2151	2152	}		
2153	2154	count = 0;		
2155	2156	do {		
2157	2158	if (++count > 10) {		
2159	2160	tmg_report_failure(me,		
2161	2162	"Unusual ramp thresholds: tmg_find_sample_period",		
2163	2164	return(FAILURE);		
2165	2166	}		
2167	2168	if (! (sp[aramp][aramp][0] = tmg_find_sample_period(aramp, aramp, t17,		
2169	2170	(long)st[aramp][aramp][0]))		
2171	2172	{		
2173	2174	tmg_report_failure(me,		
2175	2176	"Can't find new low period: tmg_find_sample_period",		
2177	2178	return(FAILURE);		
2179	2180	}		
2181	2182	st[aramp][aramp][sopoints-1] =		
2183	2184	tmg_predict_threshold(sp[aramp][aramp][0] * (sopoints + 1) / 2,		
2185	2186	slope, offset);		
2187	2188	if (--t17 < cal->Edge7MinThresh(aramp))		
2189	2190	{		
2191	2192	tmg_report_failure(me, "No allowable setting for Edge 7 Threshold");		
2193	2194	return FAILURE;		
2195	2196	}		
2197	2198	} while (st[aramp][aramp][sopoints-1] > 251);		
2199	2200	tmg_set_slot_count(sopoints + 1);		
2201	2202	if (! (sp[aramp][aramp][sopoints-1] =		
2203	2204	tmg_find_precise_sample_period(aramp, aramp, t17,		
2205	2206	(long)st[aramp][aramp][sopoints-1], (long)(sp[aramp][aramp][0]		
2207	2208	/ 2), (long)(sopoints + 1)))		
2209	2210	{		
2211	2212	tmg_report_failure(me,		
2213	2214	"Can't find high period: tmg_find_precise_sample_period",		
2215	2216	tmg_set_slot_count(2);		
2217	2218	return(FAILURE);		
2219	2220	}		
2221	2222	/* do rest of points */		
2223	2224	for(point = 1; point < sopoints - 1; point++)		
2225	2226	{		
2227	2228	st[aramp][aramp][point] = tmg_predict_threshold(sp[aramp][aramp][0]		
2229	2230	* (point+2) / 2, slope, offset);		
2231	2232	}		
2233	2234	if (! (sp[aramp][aramp][point] =		
2235	2236	tmg_find_precise_sample_period(aramp, aramp, t17,		
2237	2238	(long)st[aramp][aramp][point], (long)(sp[aramp][aramp][0]		
2239	2240	/ 2), (long)(point + 2)))		
2241	2242	{		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE	5/23/89	PAGE #
TIME	4:41:30 pm	19/166

```

LINE #          SOURCE TEXT
2161      tmg_report_failure(me,
2162      "can't find mid period: tmg_find_precise_period");
2163      tmg_set_slot_count(2);
2164      return(FAILURE);
2165  }
2166  }
2167  tmg_set_slot_count(2);
2168  /* calculate slope and offset and linearity(earmp, 71, cal,
2169  (long*)at(earmp)[aramp], (long*)ap(earmp)[aramp], soproints, aramp)
2170  == SUCCESS > 0 : 1,
2171  *errors == temperatures,
2172  returns (temperatures ? FAILURE : SUCCESS);
2173  }
2174  }
2175  }
2176  tmg_measure_sample_offset_only(earmp, aramp, cal, mode)
2177  register long aramp, aramp;
2178  register struct CALIB *cal;
2179  int mode;
2180  {
2181      static char me[] = "tmg_measure_sample_offset_only";
2182      long period;
2183      long t7, ts;
2184      long threshold;
2185      long offset;
2186      /*
2187      #ifdef DIAGS
2188      if (tmg_cal_debug & 0x100)
2189          lm_message("tmg_measure_sample_offset_only(td,td,tx,td)\n",
2190          aramp, aramp, cal, mode);
2191      #endif DIAGS
2192      /*
2193      -- 1. if EDGE7SAMPLETRIGGERMODE, use a point on Edge7 about 40 ns out
2194      -- else don't care about Edge 7
2195      -- 2. if EDGE7SAMPLETRIGGERMODE, use a minimal point on Sample Ramp
2196      -- else use point about 40 ns out
2197      -- 3. Calculate offset using previously measured slopes
2198      -- 4. Store results in calibration structure
2199      */
2200      if(mode == EDGE7SAMPLETRIGGERMODE)
2201      {
2202          t7 = Measure_edge7_points[earmp].low;
2203          ts = Measure_sample_points[aramp].low;
2204          if (!((period = tmg_find_sample_period(earmp, aramp, t7, ts))) {
2205              tmg_report_failure(me,
2206              "can't find low period: tmg_find_sample_period");
2207              return(FAILURE);
2208          }
2209          offset = (long)period
2210          - ((long)(cal->Edge7Slope[earmp] * t7)
2211          - (long)(cal->SampleSlope[aramp] * ts));
2212          threshold = (40000L - (long)offset
2213          - (long)(cal->Edge7Slope[earmp] * t7))
2214          / (long)(cal->SampleSlope[aramp]);
2215          if(tmg_cal_debug & 0x100)
2216              lm_message("predict offset = %ld, threshold = %ld\n",
2217              offset, threshold);
2218          if(threshold < cal->SampleMinThresh[aramp])
2219              threshold = cal->SampleMinThresh[aramp];
2220          if(threshold > 250)
2221              threshold = 255;
2222          ts = threshold;
2223          if (!((period = tmg_find_sample_period(earmp, aramp, t7, ts))) {
2224              tmg_report_failure(me,
2225              "can't find real low period: tmg_find_sample_period");
2226              return(FAILURE);
2227          }
2228          cal->SampleOffset[aramp][aramp] = (long)period
2229          - ((long)(cal->Edge7Slope[earmp] * t7)
2230          - (long)(cal->SampleSlope[aramp] * ts));
2231      }
2232      else if(mode == EARLYSAMPLETRIGGERMODE)
2233      {
2234          tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE;
2235          t7 = 255; /* have to keep Edge 7 quiet! */
2236          ts = Measure_early_sample_points[aramp];
2237          if (!((period = tmg_find_sample_period(01, aramp, t7, ts))) {
2238              tmg_report_failure(me, "can't find low period: tmg_find_sample_period",
2239              early mode");
2240              tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2241              return(FAILURE);
2242          }
2243          offset = (long)period
2244          - ((long)(cal->SampleSlope[aramp] * (long)ts
2245          - (long)(cal->SampleSlope[aramp] * (long)ts));
2246          if(threshold < cal->SampleMinThresh[aramp])
2247              threshold = cal->SampleMinThresh[aramp];
2248          if(threshold > 250)
2249              threshold = 255;
2250          ts = threshold;
2251          if (!((period = tmg_find_sample_period(01, aramp, t7, ts))) {
2252              tmg_report_failure(me, "can't find real low period: \
2253              tmg_find_sample_period, early mode");
2254              tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2255              return(FAILURE);
2256          }
2257          cal->EarlySampleOffset[aramp] = (long)period
2258          - ((long)(cal->SampleSlope[aramp] * (long)ts
2259          - (long)(cal->SampleSlope[aramp] * (long)ts));
2260          tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
2261      }
2262      else
2263      {
2264          lm_error("Fussy mode passed to tmg_measure_sample_offset_only\n");
2265          return FAILURE;
2266      }
2267      return SUCCESS;
2268  }
2269  }
2270  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE 5/23/89  
TIME 4:41:30 pm

PAGE #  
20/167

```

LINE # SOURCE TEXT
2281 tmg_measure_edge7_ramp(ramp, cal);
2282 register long ramp;
2283 register struct CALIB *cal;
2284 {
2285     static char me[] = "tmg_measure_edge7_ramp";
2286     long nopooints, count;
2287     int point;
2288     register long slope, offset, dt, dp;
2289     char buffer[80];
2290     #ifdef DIAGS
2291         if (tmg_cal_debug & 0x200)
2292             lm_message("tmg_measure_edge7_ramp(%d,%x)\n", ramp, cal);
2293     #endif
2294     /*
2295      * 1. Find a point above but near the minimum threshold and longer
2296      *    than the magic chip dead time.
2297      * 2. Find a point high on the ramp.
2298      * 3. Calculate the slope and offset of that line.
2299      * 3a. Use slope and offset to predict points for fine calibration.
2300      * 3b. Results of fine calibration are used for slope and offset.
2301      * 4. Find a point approx midway between the first two points.
2302      * 5. Compute a linearity value.
2303      *
2304      * Slope will be stored as a long. The units are v/s (or uv/us).
2305      */
2306     if (ramp == 0)
2307         nopooints = 3;
2308     else
2309         nopooints = 7;
2310
2311     tmg_set_slot_count(2);
2312     t[ramp][6][0] = Measure_edge7_points(ramp).low;
2313     if (! (p[ramp][6][0] = tmg_find_sample_period(ramp, 01,
2314         (long)t[ramp][6][0], 101)))
2315     {
2316         tmg_report_failure(me,
2317             "can't find low period: tmg_find_sample_period");
2318         return (FAILURE);
2319     }
2320     t[ramp][6][2] = Measure_edge7_points(ramp).high;
2321     if (! (p[ramp][6][2] = tmg_find_sample_period(ramp, 01,
2322         (long)t[ramp][6][2], 101)))
2323     {
2324         tmg_report_failure(me,
2325             "can't find high period: tmg_find_sample_period");
2326         return (FAILURE);
2327     }
2328     if ((dt = t[ramp][6][2] - t[ramp][6][0]) == 0) {
2329         lm_error("Infinite edge 7 ramp slope\n");
2330         return (FAILURE);
2331     }
2332     if ((dp = p[ramp][6][2] - p[ramp][6][0]) == 0) {
2333         lm_error("Zero edge 7 ramp slope\n");
2334         return (FAILURE);
2335     }
2336     /* slopes are normally 300, 2000, 10000, and 50000 */
2337     slope = dp/dt;
2338     if ((slope < 50) || (slope > 50000)) {
2339         lm_error("Illegal edge 7 ramp slope %d ps/thr\n", slope);
2340         return (FAILURE);
2341     }
2342     offset = p[ramp][6][0] - slope*t[ramp][6][0];
2343     p[ramp][6][0] = tmg_predict_period(Measure_edge7_points(ramp).high,
2344         slope, offset) / nopooints;
2345     t[ramp][6][0] = tmg_predict_threshold(p[ramp][6][0], slope, offset);
2346     if (t[ramp][6][0] < cal->Edge7MinThresh(ramp))
2347     {
2348         sprintf(buffer, "Can't fit 2d periods in Edge7 ramp %d", nopooints, ramp);
2349         tmg_report_failure(me, buffer);
2350         return (FAILURE);
2351     }
2352     count = 0;
2353     do {
2354         if (++count > 10) {
2355             tmg_report_failure(me,
2356                 "Unusual ramp thresholds: tmg_find_sample_period");
2357             return (FAILURE);
2358         }
2359         if (! (p[ramp][6][0] = tmg_find_sample_period(ramp, 01,
2360             (long)t[ramp][6][0], 101)))
2361         {
2362             tmg_report_failure(me,
2363                 "can't find low period: tmg_find_sample_period");
2364             return (FAILURE);
2365         }
2366         t[ramp][6][nopooints-1] =
2367             tmg_predict_threshold(p[ramp][6][0] * nopooints,
2368                 slope, offset);
2369         if (! (t[ramp][6][0] < cal->Edge7MinThresh(ramp)))
2370         {
2371             tmg_report_failure(me, "No allowable setting for Edge 7 Threshold");
2372             return (FAILURE);
2373         }
2374     } while (t[ramp][6][nopooints-1] > 251);
2375     tmg_set_slot_count(nopooints + 1);
2376     if (! (p[ramp][6][nopooints-1] =
2377         tmg_find_precise_sample_period(ramp, 01,
2378             (long)t[ramp][6][nopooints-1], 101, (long)p[ramp][6][0],
2379             (long)nopooints)))
2380     {
2381         tmg_report_failure(me,
2382             "can't find high period: tmg_find_precise_sample_period");
2383         tmg_set_slot_count(2);
2384         return (FAILURE);
2385     }
2386     /* do rest of points */
2387     for (point = 1; point < nopooints - 1; point++)
2388     {
2389         t[ramp][6][point] = tmg_predict_threshold(p[ramp][6][0]
2390             * (point+1), slope, offset);

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

21/168

```

LINE # SOURCE TEXT
2401
2402 if (!p(ramp)[6][point] =
2403     tmg_find_precise_sample_period(ramp, 01,
2404     (long)t(ramp)[6][point], 101, (long)p(ramp)[6][0],
2405     (long)(point - 1)))
2406 {
2407     tmg_report_failure(ma,
2408     "can't find mid period: tmg_find_precise_period"),
2409     tmg_set_slot_count(2),
2410     return(FAILURE);
2411 }
2412
2413 tmg_set_slot_count(2);
2414 return calculate_slope_and_offset_and_linearity(ramp, 61, cal,
2415     (long)t(ramp)[6], (long)p(ramp)[6], npoints, 01);
2416 }
2417
2418 tmg_complete_calib_structure(calptr)
2419 struct CALIB *calptr;
2420 {
2421     long eramp, aramp, min_edge_slope, max_edge_slope, ave_slope;
2422     long max_min_delay, min_max_delay;
2423     long edge;
2424     long temp;
2425
2426     /* Figure out EdgeMinDelay[] and EdgeMaxDelay[]
2427     /* average various edge slopes along with Edge7
2428
2429     eramp = 0;
2430     {
2431         max_edge_slope = 0;
2432         min_edge_slope = 1000000;
2433         max_min_delay = 0;
2434         min_max_delay = 100000000;
2435         ave_slope = calptr->Edge7Slope(eramp);
2436         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2437         {
2438             if(calptr->EdgeSlope(eramp)[edge] > max_edge_slope)
2439                 max_edge_slope = calptr->EdgeSlope(eramp)[edge];
2440             if(calptr->EdgeSlope(eramp)[edge] < min_edge_slope)
2441                 min_edge_slope = calptr->EdgeSlope(eramp)[edge];
2442             ave_slope += calptr->EdgeSlope(eramp)[edge];
2443         }
2444         ave_slope /= NUMBER_OF_EDGES + 1;
2445         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2446         {
2447             calptr->EdgeSlope(eramp)[edge] = ave_slope;
2448             /* min thresh + 2 for thermal tail effects
2449             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2450             + calptr->EdgeOffset(eramp)[edge];
2451             if(max_min_delay < temp)
2452                 max_min_delay = temp;
2453             temp = (255 - END_DEAD_TIME_FUDGE) * (long)min_edge_slope
2454             + calptr->EdgeOffset(eramp)[edge];
2455             - 2 * (long)ave_slope; /* jitter */
2456             if(min_max_delay > temp)
2457                 min_max_delay = temp;
2458         }
2459         calptr->Edge7Slope(eramp) = ave_slope;
2460         calptr->EdgeMinDelay(eramp) = max_min_delay;
2461         calptr->EdgeMaxDelay(eramp) = min_max_delay;
2462     }
2463     for(eramp = 1; eramp < NUMBER_OF_ERAMPS; eramp++)
2464     {
2465         max_edge_slope = 0;
2466         min_edge_slope = 1000000;
2467         max_min_delay = 0;
2468         min_max_delay = 100000000;
2469         ave_slope = calptr->Edge7Slope(eramp);
2470         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2471         {
2472             if(calptr->EdgeSlope(eramp)[edge] > max_edge_slope)
2473                 max_edge_slope = calptr->EdgeSlope(eramp)[edge];
2474             if(calptr->EdgeSlope(eramp)[edge] < min_edge_slope)
2475                 min_edge_slope = calptr->EdgeSlope(eramp)[edge];
2476             ave_slope += calptr->EdgeSlope(eramp)[edge];
2477         }
2478         ave_slope /= NUMBER_OF_EDGES + 1;
2479         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2480         {
2481             calptr->EdgeSlope(eramp)[edge] = ave_slope;
2482             /* min thresh + 2 for thermal tail effects
2483             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2484             + calptr->EdgeOffset(eramp)[edge];
2485             if(max_min_delay < temp)
2486                 max_min_delay = temp;
2487             temp = (255 - END_DEAD_TIME_FUDGE) * (long)min_edge_slope
2488             + calptr->EdgeOffset(eramp)[edge];
2489             - 2 * (long)ave_slope; /* jitter */
2490             if(min_max_delay > temp)
2491                 min_max_delay = temp;
2492         }
2493         calptr->Edge7Slope(eramp) = ave_slope;
2494         calptr->EdgeMinDelay(eramp) = max_min_delay;
2495         calptr->EdgeMaxDelay(eramp) = min_max_delay;
2496     }
2497     /* Now we have to account for cases where a small step in the ramp
2498     /* causes a large negative offset that fools us into thinking
2499     /* the minimum delay is small. The indication of this is when:
2500     /* a slower ramp has a smaller minimum delay than a faster ramp
2501     if(calptr->EdgeMinDelay(eramp) < calptr->EdgeMinDelay(eramp-1))
2502     {
2503         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2504         {
2505             /* increase the minimum threshold as necessary
2506             while(((long)(calptr->EdgeMinThresh(eramp)[edge] + 2)
2507             + calptr->EdgeSlope(eramp)[edge]
2508             + calptr->EdgeOffset(eramp)[edge])
2509             < calptr->EdgeMinDelay(eramp-1))
2510                 ++(calptr->EdgeMinThresh(eramp)[edge]);
2511         }
2512         /* recompute minimum delays
2513         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2514         {
2515             temp = (long)(calptr->EdgeMinThresh(eramp)[edge] + 2) * max_edge_slope
2516             + calptr->EdgeOffset(eramp)[edge];
2517             if(max_min_delay < temp)
2518                 max_min_delay = temp;
2519         }
2520         calptr->EdgeMinDelay(eramp) = max_min_delay;
2521     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

22/169

```

2521 }
2522
2523 /* now figure out SampleMinDelay */
2524 for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2525 {
2526     max_min_delay = 0;
2527     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2528     {
2529         /* min thresh + 2 for thermal tail effects
2530          * calptr->Edge7MinThresh[aramp] + 2) */
2531         temp = (long)((calptr->Edge7MinThresh[aramp] + 2)
2532             + (long)(calptr->SampleMinThresh[aramp] * calptr->SampleSlope[aramp])
2533             + calptr->SampleOffset[aramp]);
2534         if(tmg_cal_debug & 0x00000000)
2535         {
2536             lm_message("\taramp %ld, aramp %ld\n", aramp, aramp);
2537             lm_message("\tEdge7MinTh = %ld, Edge7Slope = %ld\n",
2538                 calptr->Edge7MinThresh[aramp], calptr->Edge7Slope[aramp]);
2539             lm_message("\tSampleMinTh = %ld, SampleSlope = %ld\n",
2540                 calptr->SampleMinThresh[aramp], calptr->SampleSlope[aramp]);
2541             lm_message("\ttemp = %ld\n", temp);
2542         }
2543         if(max_min_delay < temp)
2544             max_min_delay = temp;
2545     }
2546     calptr->SampleMinDelay[aramp] = max_min_delay;
2547     if(tmg_cal_debug & 0x00000000)
2548         lm_message("\tSampleMinDelay = %ld\n", max_min_delay);
2549 }
2550
2551 for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2552 {
2553     calptr->EarlySampleMinDelay[aramp] = (long)(calptr->SampleMinThresh[aramp]
2554         + calptr->SampleSlope[aramp]
2555         + calptr->EarlySampleOffset[aramp]
2556         + (2551 - END_DEAD_TIME_FUDGE)
2557         + (long)calptr->SampleSlope[aramp]
2558         + calptr->EarlySampleOffset[aramp]);
2559 }
2560
2561 return SUCCESS;
2562 }
2563
2564 /*
2565 * tmg_print_calib_structures(calptr)
2566 * This routine prints the relevant stuff contained within
2567 * a calibration structure.
2568 * Inputs: pointer to calib structure
2569 * Outputs: always returns SUCCESS
2570 */
2571
2572 tmg_print_calib_structures(calptr)
2573 struct CALIB *calptr;
2574 {
2575     long aramp, aramp;
2576     long edge;
2577
2578     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2579     {
2580         lm_message("Edge Ramp %d:\n", aramp);
2581         lm_message("\tMinimum delay = %d ps\n", calptr->EdgeMinDelay[aramp]);
2582         lm_message("\tMaximum delay = %d ps\n", calptr->EdgeMaxDelay[aramp]);
2583         lm_message("\tslope = %d ps/threshold\n", calptr->EdgeSlope[aramp][0]);
2584         lm_message("\tEdge %d to %d to %d to %d to %d to %d to %d\n",
2585             for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2586             {
2587                 lm_message("\td", calptr->EdgeOffset[aramp][edge]);
2588                 lm_message("\tlinearity");
2589             }
2590             for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2591             {
2592                 lm_message("\td", calptr->EdgeLinearity[aramp][edge]);
2593                 lm_message("\n");
2594             }
2595     }
2596     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2597     {
2598         lm_message("\tEdge Ramp %d:\n", aramp);
2599         lm_message("\tSample Minimum Delay = %d\n", calptr->SampleMinDelay[aramp]);
2600         lm_message("\tSample Ramp %d to %d to %d to %d to %d to %d to %d\n",
2601             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2602             {
2603                 lm_message("\td", calptr->SampleOffset[aramp][aramp]);
2604                 lm_message("\n");
2605             }
2606         lm_message("\tSample Ramp %d to %d to %d to %d to %d to %d to %d\n",
2607             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2608             {
2609                 lm_message("\td", calptr->SampleLinearity[aramp]);
2610                 lm_message("\n");
2611             }
2612         lm_message("\tEarly Sample Ramp:\n");
2613         lm_message("\tSample %d to %d to %d to %d to %d to %d to %d\n",
2614             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2615             {
2616                 lm_message("\td", calptr->EarlySampleMinDelay[aramp]);
2617                 lm_message("\n");
2618             }
2619         lm_message("\tEarly Sample Offset:\n");
2620         lm_message("\tSample %d to %d to %d to %d to %d to %d to %d\n",
2621             for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2622             {
2623                 lm_message("\td", calptr->EarlySampleOffset[aramp]);
2624                 lm_message("\n");
2625             }
2626     }
2627     return SUCCESS;
2628 }
2629
2630 /*
2631 * tmg_print_data_logging_messages(calptr)
2632 * This routine prints the relevant stuff contained within
2633 * a data logging structure.
2634 * Inputs: pointer to data logging structure
2635 * Outputs: always returns SUCCESS
2636 */
2637
2638 tmg_print_data_logging_messages(calptr)
2639 struct DATA_LOG *calptr;
2640 {
2641     int aramp, edge, aramp, lineno = 0;
2642     static char leader[] = "DATALOG: ";
2643
2644     lm_message("\tstd BEGIN Time = %d\n", leader, lineno++, lm_time());
2645     for(aramp = 0; aramp < NUMBER_OF_RAMPS; aramp++)
2646     {
2647         lm_message("\tstd Edge Ramp %d\n", leader, lineno++, aramp);
2648         lm_message("\tstd Minimum Thresholds", leader, lineno++);
2649         for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2650             lm_message("\td", calptr->EdgeMinThresh[aramp][edge]);
2651         lm_message("\n");
2652         lm_message("\tstd Minimum Delay %d\n", leader, lineno++,
2653             calptr->EdgeMinDelay[aramp]);
2654         lm_message("\tstd Maximum Delay %d\n", leader, lineno++,
2655             calptr->EdgeMaxDelay[aramp]);
2656     }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

23/170

```

LINE # SOURCE TEXT
2641  ln_message("tstd Edge Offset", leader, lineno++);
2642  for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2643  ln_message("  td", calptr->EdgeOffset[aramp][edge]);
2644  ln_message("\n");
2645
2646  ln_message("tstd Edge Slope", leader, lineno++);
2647  for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2648  ln_message("  td", calptr->EdgeSlope[aramp][edge]);
2649  ln_message("\n");
2650
2651  ln_message("tstd Edge Linearity", leader, lineno++);
2652  for(edge = 0; edge < NUMBER_OF_EDGES; edge++)
2653  ln_message("  td", calptr->EdgeLinearity[aramp][edge]);
2654  ln_message("\n");
2655
2656  ln_message("tstd Edge 7 Minimum Threshold td\n", leader, lineno++);
2657  calptr->Edge7MinThresh[aramp];
2658
2659  ln_message("tstd Edge 7 Slope td\n", leader, lineno++);
2660  calptr->Edge7Slope[aramp];
2661
2662  ln_message("tstd Edge 7 Linearity td\n", leader, lineno++);
2663  calptr->Edge7Linearity[aramp];
2664
2665  for(aramp = 0; aramp < NUMBER_OF_ARAMP; aramp++)
2666  {
2667      ln_message("tstd Sample Ramp td\n", leader, lineno++, aramp);
2668
2669      ln_message("tstd Sample Minimum Threshold td\n", leader, lineno++,
2670      calptr->SampleMinThresh[aramp]);
2671
2672      ln_message("tstd Sample Minimum Delay td\n", leader, lineno++,
2673      calptr->SampleMinDelay[aramp]);
2674
2675      ln_message("tstd Sample Maximum Delay td\n", leader, lineno++,
2676      calptr->SampleMaxDelay[aramp]);
2677
2678      ln_message("tstd Sample Slope td\n", leader, lineno++,
2679      calptr->SampleSlope[aramp]);
2680
2681      ln_message("tstd Sample Linearity td\n", leader, lineno++,
2682      calptr->SampleLinearity[aramp]);
2683
2684      ln_message("tstd Sample Offset", leader, lineno++);
2685      for(aramp = 0; aramp < NUMBER_OF_ARAMP; aramp++)
2686      ln_message("  td", calptr->SampleOffset[aramp][aramp]);
2687      ln_message("\n");
2688
2689      ln_message("tstd Early Sample Minimum Delay td\n", leader, lineno++,
2690      calptr->EarlySampleMinDelay[aramp]);
2691
2692      ln_message("tstd Early Sample Maximum Delay td\n", leader, lineno++,
2693      calptr->EarlySampleMaxDelay[aramp]);
2694
2695      ln_message("tstd Early Sample Offset td\n", leader, lineno++,
2696      calptr->EarlySampleOffset[aramp]);
2697
2698  }
2699
2700  ln_message("tstd Delay Line Setting td\n", leader, lineno++,
2701  calptr->DelayDelay);
2702
2703  ln_message("tstd END\n", leader, lineno++);
2704
2705
2706  /*
2707  * tmg_process_and_transfer_calib_structure(temptr, realptr)
2708  *
2709  * This routine processes all the data in the temporary calibration
2710  * structure and transfers it to the real one.
2711  *
2712  * Inputs: pointers to the temporary and real calib structures
2713  * Outputs: returns SUCCESS or FAILURE
2714  */
2715  tmg_process_and_transfer_calib_structure(temptr, realptr)
2716  struct CALIB *temptr, *realptr;
2717  {
2718      int i;
2719      char *from, *to;
2720      tmg_complete_calib_structure(temptr);
2721      from = (char *)temptr;
2722      to = (char *)realptr;
2723      for(i = 0; i < sizeof(struct CALIB); i++)
2724      *to++ = *from++;
2725      return SUCCESS;
2726  }
2727
2728
2729  tmg_is_calibrated()
2730  {
2731      return (calib.CalCompleted == CALIBRATION_DONE);
2732  }
2733
2734  static long tmg_save_test_mode, tmg_save_delay_reg;
2735
2736  tmg_restore_calibration()
2737  {
2738      if (!tmg_is_calibrated()) {
2739          if (tmg_fast_calibrate() != SUCCESS)
2740              return FAILURE;
2741      } else {
2742          tmg_set_test_mode(tmg_save_test_mode);
2743          tmg_set_delay_reg(tmg_save_delay_reg);
2744      }
2745      return SUCCESS;
2746  }
2747
2748  tmg_save_calibration()
2749  {
2750      tmg_save_test_mode = tmg_get_test_mode();
2751      tmg_save_delay_reg = tmg_get_delay_reg();
2752  }
2753
2754  /*
2755  * tmg_set_timing(period, edgetime, setime, setime, spw)
2756  *
2757  * Inputs: period clock period in picoseconds
2758  *          edgetime array of 7 edge times in picoseconds
2759  *          setime sample start time in picoseconds
2760  *          setime latest expected sample start time in picoseconds

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

24/171

```

LINE # SOURCE TEXT
2761 ** (used to select sample ramp)
2762 ** sample pulse width in picoseconds
2763 **
2764
2765 tmg_set_timing(period, edgetime, astime, setime, spw)
2766 long period, edgetime, astime, setime, spw;
2767 {
2768     static char me[] = "tmg_set_timing";
2769     long startdeadtime,
2770     long enddeadtime,
2771     long thresholds[NUMBER_OF_EDGES + 1],
2772     long ramp, edge, thr, slope, offset, arange, sthreshold,
2773     long edge_limit,
2774     long sample_delta, arampmindelay, minst,
2775
2776     /* add start deadtime to andy's numbers */
2777
2778     /* select edge ramp: */
2779
2780     for (ramp = 0; ramp < NUMBER_OF_RAMPS; ++ramp) {
2781         tmg_get_ramp_dead_time(period, ramp, &startdeadtime, &enddeadtime);
2782         slope = calib.EdgesSlope[ramp][0];
2783         offset = calib.EdgesOffset[ramp][0];
2784         thr = tmg_predict_threshold(period - enddeadtime, slope, offset);
2785         if ((thr >= calib.EdgesMinThresh[ramp][0]) && (thr < 255))
2786             break;
2787     }
2788
2789     if (ramp >= NUMBER_OF_RAMPS) {
2790         ln_error("Requested period out of range\n", me);
2791         return FAILURE;
2792     }
2793
2794     /* Now we have selected our edge ramp, calculate thresholds: */
2795
2796     /* Now, all edge settings are relative to time 0 (at minst): */
2797     for (edge = 0; edge < NUMBER_OF_EDGES; edge++) {
2798         edge_limit = tmg_predict_period(thr,
2799             calib.EdgesSlope[ramp][edge], calib.EdgesOffset[ramp][edge]);
2800         thresholds[edge] = tmg_predict_threshold(edgetime[edge],
2801             calib.EdgesSlope[ramp][edge], 0) + calib.EdgesMinThresh[ramp][edge];
2802         if (edgetime[edge] > edge_limit) {
2803             if (tmg_cal_debug & 0x4000000) {
2804                 ln_warning("ts:Edge %d (slope=%d thr %d) too large for ramp %d\n",
2805                     me, edge, edgetime[edge], edge_limit, thresholds[edge], ramp);
2806                 thresholds[edge] = 255; /* shut the sucker off! */
2807             }
2808         }
2809     }
2810
2811     /* select sample mode: */
2812     tmg_ptr->sample_mode = EDGE7AMPLETRIGGERMODE;
2813
2814     /* select sample range: */
2815     sample_delta = setime - astime;
2816     for (arange = 0; arange < NUMBER_OF_RAMPS; ++arange) {
2817         if (sample_delta <
2818             (255 - END_DEAD_TIME_FUDGE - calib.SampleMinThresh[arange])
2819             * calib.SampleSlope[arange])
2820             break;
2821     }
2822     if (arange >= NUMBER_OF_RAMPS) {
2823         ln_error("sample time too big\n", me);
2824         return FAILURE;
2825     }
2826     sthreshold = calib.SampleMinThresh[arange];
2827
2828     /* now calculate the minimum legal sample time: */
2829     arampmindelay = tmg_predict_period(calib.SampleMinThresh[arange],
2830         calib.SampleSlope[arange], calib.SampleOffset[ramp][arange]);
2831     minst = arampmindelay
2832         + tmg_predict_period(calib.Edges7MinThresh[ramp],
2833             calib.Edges7Slope[ramp], 0);
2834
2835     /* select edge 7 threshold */
2836     thresholds[6] = tmg_predict_threshold(astime + minst,
2837         calib.Edges7Slope[ramp], arampmindelay);
2838
2839     /* set clock period */
2840     if (tmg_set_period(ramp, &period) != SUCCESS) {
2841         tmg_report_failure(me, "tmg_set_period");
2842         return FAILURE;
2843     }
2844
2845     /* set edges and edge ramp */
2846     if (ln_tmg_set_ramp_edge_settings(ramp, thresholds) != SUCCESS) {
2847         tmg_report_failure(me, "ln_tmg_set_ramp_edge_settings");
2848         return FAILURE;
2849     }
2850
2851     /* set sample range and threshold */
2852     if (ln_tmg_set_rising_sample(arange, sthreshold) != SUCCESS) {
2853         tmg_report_failure(me, "ln_tmg_set_rising_sample");
2854         return FAILURE;
2855     }
2856
2857     /* set sample pulse width */
2858     if (tmg_set_sample_width(period, slope, spw) != SUCCESS) {
2859         tmg_report_failure(me, "tmg_set_sample_width");
2860         return FAILURE;
2861     }
2862
2863     if (tmg_cal_debug & 0x4000000) {
2864         ln_message("ts:\n", me);
2865         ln_message("\t period = %d\n", period);
2866         ln_message("\t ramp = %d\n", ramp);
2867         for (edge = 0; edge <= NUMBER_OF_EDGES; edge++) {
2868             ln_message("\t Edge %d threshold = %d\n", edge, thresholds[edge]);
2869         }
2870         ln_message("\t sample range = %d\n", arange);
2871         ln_message("\t sample threshold = %d\n", sthreshold);
2872         ln_message("\t sample pulse width register = %d\n",
2873             tmg_get_sample_width_reg(sample_width));
2874     }
2875
2876     return SUCCESS;
2877 }
2878
2879 /* tmg_fast_calibrate() uses conservative default values for slopes */
2880

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 25/172
LINE #		SOURCE TEXT		
2881	2882	/* etc, then really measure and sets the proper delay. */		
2883	2884	tmg_fast_calibrate()		
2885	2886	{		
2887	2888	static char me[] = "tmg_fast_calibrate";		
2889	2890	static struct CALIB default_calib;		
2891	2892	static long fast_cal_delay = 0;		
2893	2894	long errors;		
2895	2896	register long edge;		
2897	2898	bcopy(&default_calib, &calib, sizeof(calib));		
2899	2900	tmg_set_slot_count(2);		
2901	2902	tmg_set_test_mode(1);		
2903	2904	{		
2905	2906	if (!fast_cal_delay) {		
2907	2908	tmg_set_delay(14);		
2909	2910	/* delay = 5ns + 14ns */		
2911	2912	if (tmg_measure_ramp(0l, 0l, &calib, &errors) != SUCCESS) {		
2913	2914	tmg_report_failure(me, "tmg_measure_ramp(0,0)");		
2915	2916	tmg_set_slot_count(1);		
2917	2918	tmg_set_test_mode(0);		
2919	2920	return FAILURE;		
2921	2922	for (edge = 1; edge < NUMBER_OF_EDGES; ++edge) {		
2923	2924	calib.EdgeSlope[0][edge] = calib.EdgeSlope[0][0];		
2925	2926	calib.EdgeOffset[0][edge] = calib.EdgeOffset[0][0];		
2927	2928	}		
2929	2930	if (tmg_calibrate_delay(&calib) != SUCCESS) {		
2931	2932	tmg_report_failure(me, "tmg_calibrate_delay");		
2933	2934	tmg_set_slot_count(1);		
2935	2936	tmg_set_test_mode(0);		
2937	2938	return FAILURE;		
2939	2940	fast_cal_delay = tmg_get_delay();		
2941	2942	} else {		
2943	2944	tmg_set_delay(fast_cal_delay);		
2945	2946	tmg_set_slot_count(1);		
2947	2948	tmg_set_test_mode(0);		
2949	2950	return SUCCESS;		
2951	2952	}		
2953	2954	/* The following functions were put here for Robert's benefit */		
2955	2956	int tmg_effclear(void)		
2957	2958	{		
2959	2960	/* Clears edge and sample flip flops		
2961	2962	Inputs: none		
2963	2964	Returns: SUCCESS or FAILURE		
2965	2966	int tmg_effclear() /* clear the cal flip flops */		
2967	2968	{		
2969	2970	int returncode;		
2971	2972	tmgptr->as_if_clearl = 0;		
2973	2974	if(tmgptr->edge_cal & calif) /* see if any still set */		
2975	2976	{		
2977	2978	lm_error("Could not clear cal flip flops\n");		
2979	2980	returncode = FAILURE;		
2981	2982	}		
2983	2984	else		
2985	2986	returncode = SUCCESS;		
2987	2988	tmgptr->as_if_clearl = 1;		
2989	2990	return(returncode);		
2991	2992	}		
2993	2994	/* int tmg_play(timeout)		
2995	2996	{		
2997	2998	/* Initiates pattern presentation		
2999	3000	Inputs: none		
3001	3002	Returns: SUCCESS or FAILURE		
3003	3004	int tmg_play(timeout) /* start a presentation */		
3005	3006	{		
3007	3008	long timeout;		
3009	3010	{		
3011	3012	if(tmg_initiate_play() != SUCCESS)		
3013	3014	return(FAILURE);		
3015	3016	if(tmg_complete_play(timeout) != SUCCESS)		
3017	3018	return(FAILURE);		
3019	3020	else		
3021	3022	return(SUCCESS);		
3023	3024	}		
3025	3026	}		
3027	3028	/* int tmg_initiate_play(void)		
3029	3030	{		
3031	3032	/* This routine turns clocks on and starts a presentation.		
3033	3034	Inputs: none		
3035	3036	Outputs: function returns SUCCESS or FAILURE		
3037	3038	int tmg_initiate_play()		
3039	3040	{		
3041	3042	if(tmg_clockoff() != SUCCESS)		
3043	3044	{		
3045	3046	lm_error("tmg_initiate_play: tmg_clockoff returned error\n");		
3047	3048	return(FAILURE);		
3049	3050	}		
3051	3052	if(tmg_clockon() != SUCCESS)		
3053	3054	{		
3055	3056	lm_error("play: tmg_clockon returned error\n");		
3057	3058	return(FAILURE);		
3059	3060	}		
3061	3062	tmgptr->pattern_intr_enable = 0; /* just in case */		
3063	3064	tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */		
3065	3066	tmgptr->start_pattern_play = 1; /* start presentation */		
3067	3068	return(SUCCESS);		
3069	3070	}		
3071	3072	}		
3073	3074	/* int tmg_complete_play(long timeout)		
3075	3076	{		
3077	3078	/* This function waits for a presentation to complete by		
3079	3080	waiting for the EOP interrupt from the Timing Generator.		
3081	3082	}		

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_cal.c

DATE

5/23/89

PAGE #

TIME

4:41:30 pm

26/173

```

LINE # SOURCE TEXT
3001 * Inputs: timeout is ms
3002 * Outputs: function returns SUCCESS or FAILURE
3003 */
3004 int tmg_complete_play(timeout)
3005 {
3006     long start;
3007     int returncode = SUCCESS;
3008     start = lm_time();
3009     #ifndef DIAGS
3010     while (!cpu_tmg_interrupt_status())
3011     #else
3012     while (!TMC_INT) /* wait for interrupt */
3013     #endif
3014     {
3015         if ((lm_time() - start) > timeout)
3016         {
3017             lm_error("tmg_complete_play: timeout waiting for access mode\n");
3018             returncode = FAILURE;
3019             break;
3020         }
3021         tmgptr->pattern_intr_enable = 0; /* clear EOP interrupt */
3022         if (tmg_clockoff() != SUCCESS)
3023             returncode = FAILURE;
3024         return(returncode);
3025     }
3026 }
3027
3028 /* int tmg_clockoff(void)
3029 * Turns clock off
3030 * Input: nothing
3031 * Output: SUCCESS or FAILURE
3032 */
3033 int tmg_clockoff()
3034 {
3035     long start;
3036     tmgptr->clock_enable = 0;
3037     start = lm_time();
3038     while (tmgptr->clock_on)
3039     {
3040         if ((lm_time() - start) > 10)
3041         {
3042             if (tmgptr->clock_select > 2) /* must be external clock */
3043                 tmgptr->clock_sync_clear1 = 0;
3044             break;
3045         }
3046         if (tmgptr->clock_on)
3047             return(FAILURE);
3048         tmgptr->clock_sync_clear1 = 0;
3049         return(SUCCESS);
3050     }
3051 }
3052
3053 /* int tmg_clockon(void)
3054 * Turns clock on
3055 * Input: nothing
3056 * Output: SUCCESS or FAILURE
3057 */
3058 int tmg_clockon()
3059 {
3060     long start;
3061     int playmode;
3062     playmode = tmgptr->backplane_mode;
3063     tmgptr->clock_enable = 0;
3064     tmgptr->clock_sync_clear1 = 1;
3065     tmgptr->clock_enable = 1;
3066     /* There is one weird condition that if the state machine is
3067     /* caught in play mode and the TM is reset, when the clock
3068     /* turns on, PLAY goes low and turns it off again.
3069     if (playmode)
3070     {
3071         start = lm_time();
3072         while (!tmgptr->clock_on)
3073             if ((lm_time() - start) > 10)
3074             {
3075                 tmgptr->clock_enable = 0;
3076                 tmgptr->clock_enable = 1;
3077                 break;
3078             }
3079     }
3080     start = lm_time();
3081     while (!tmgptr->clock_on)
3082     {
3083         if ((lm_time() - start) > 10)
3084         {
3085             lm_error("tmg_clockon: did not turn on\n");
3086             return(FAILURE);
3087         }
3088     }
3089     return(SUCCESS);
3090 }
3091
3092 tmg_set_defaults()
3093 {
3094     *REG(0) = 0x10; /* not ORESET, not RESET_OUT */
3095     *REG(1) = 0x00; /* 25 MHz
3096     *REG(2) = 0x2c; /* Select divide by 2
3097     *REG(3) = 0x0ff; /* Select fast ramps, no lanes
3098     *REG(4) = 0x00; /* Test mode 1, 18ns delay
3099     *REG(5) = 0x62; /* 18ns dump, ext clock
3100     *REG(6) = 0x07; /* 16 clock sample width
3101     *REG(7) = 0x07; /* minimum threshold EDGE[0]
3102     *REG(16) = 0x07; /* EDGE[1]
3103     *REG(17) = 0x07; /* EDGE[2]
3104     *REG(18) = 0x07; /* EDGE[3]
3105     *REG(19) = 0x07; /* EDGE[4]
3106     *REG(20) = 0x07; /* EDGE[5]
3107     *REG(21) = 0x07; /* SAMPLE Trigger
3108     *REG(22) = 0x07; /* SAMPLE
3109     *REG(23) = 0x07;
3110 }
3111
3112 int tmg_reset(do_bp_reset)
3113 {
3114     /* This function performs a power up initialization sequence.
3115     /* Inputs: do_bp_reset = TRUE if backplane reset desired, else FALSE
3116     /* Outputs: function returns SUCCESS or FAILURE
3117 }
3118
3119
3120

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_cal.c

DATE 5/23/89  
TIME 4:41:30 pm

PAGE #  
27/174

```

LINE # SOURCE TEXT
3121 int
3122 tmg_reset(do_bp_reset)
3123 int do_bp_reset;
3124 {
3125     int returncode = SUCCESS;
3126
3127     #ifdef DIAGS
3128     if(!Host)
3129         return(returncode);
3130
3131     if (do_bp_reset) pac_info_init();
3132     #endif DIAGS
3133     tmg_set_defaults();
3134     tmgptr->tmg_resetl = 0;
3135     if (do_bp_reset == TRUE)
3136         tmgptr->backplane_resetl = 0;
3137
3138     lm_delay(1); /* wait for PLL to lock */
3139
3140     /* some synchronous magic here... */
3141     if(tmg_clockoff() != SUCCESS)
3142     {
3143         lm_error("Reset: cannot turn clock off.\n");
3144         returncode = FAILURE;
3145     }
3146     if(tmg_clockon() != SUCCESS)
3147     {
3148         lm_error("Reset-1: cannot turn clock on.\n");
3149         returncode = FAILURE;
3150     }
3151     if(tmg_clockoff() != SUCCESS)
3152     {
3153         lm_error("Reset: cannot turn clock off.\n");
3154         returncode = FAILURE;
3155     }
3156
3157     tmgptr->tmg_resetl = 1; /* remove TMG reset */
3158     if(tmg_play((long)TIMEOUT) != SUCCESS)
3159     {
3160         lm_error("Reset: cannot play.\n");
3161         returncode = FAILURE;
3162     }
3163
3164     tmg_set_test_mode(0);
3165
3166     if (do_bp_reset == TRUE)
3167     {
3168         if(tmg_clockon() != SUCCESS)
3169         {
3170             lm_error("Reset-2: cannot turn clock on.\n");
3171             returncode = FAILURE;
3172         }
3173         lm_delay(5);
3174         if(tmg_clockoff() != SUCCESS)
3175         {
3176             lm_error("Reset: cannot turn clock off.\n");
3177             returncode = FAILURE;
3178         }
3179         lm_delay(5);
3180         tmgptr->backplane_resetl = 1; /* remove backplane reset */
3181
3182         /* This next section of code checks for backplane errors
3183          * and attempts to remove them by probing the PACs in every
3184          * lane (whether they are there or not). This is because it
3185          * is possible for the TMC to have a bogus refresh error
3186          * after the reset is de-asserted. Probing the error register
3187          * will remove this error. If errors still exist after the
3188          * probing takes place, the function returns FAILURE.
3189          */
3190         if(tmgptr->lane_intr != 0)
3191         {
3192             (void)lm_write_probe(0x0c1002801, 01);
3193             (void)lm_write_probe(0x0c1002801, 01);
3194             (void)lm_write_probe(0x0c1002801, 01);
3195             (void)lm_write_probe(0x0c1002801, 01);
3196         }
3197     }
3198
3199     if(bp_mode() == PLAY_MODE)
3200     {
3201         (void)lm_error("Backplane is still in play mode after TMC init.\n");
3202         return FAILURE;
3203     }
3204
3205     #ifdef DIAGS
3206     /* Check for backplane errors */
3207     if(diag_clear_errors() != SUCCESS)
3208     {
3209         return FAILURE;
3210     }
3211     #endif
3212     return returncode;
3213 }
3214
3215 diag_tmg_reset(do_bp_reset)
3216 int do_bp_reset;
3217 {
3218     int returncode;
3219     static long edgetime[] = {
3220         5000, /* edge 0 */
3221         5000, /* edge 1 */
3222         5000, /* edge 2 */
3223         15000, /* edge 3 */
3224         5000, /* edge 4 */
3225         0, /* edge 5 */
3226     };
3227
3228     returncode = tmg_reset(do_bp_reset);
3229
3230     if (tmg_restore_calibration() != SUCCESS) {
3231         (void)lm_error("Unable to restore calibration\n");
3232         return FAILURE;
3233     }
3234     tmg_set_slot_count(1);
3235     if (tmg_set_timing(1000001, edgetime, 01, 1000001, 25000001) != SUCCESS) {
3236         lm_error("Could not set edges\n");
3237         returncode = FAILURE;
3238     }
3239 }
3240

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_cal.c	DATE 5/23/89	PAGE # 28/175
LINE #		SOURCE TEXT		
3241		return(retaracode);		
3242		}		
3243		tmg_report_failure(called_from, failing_func)		
3244		char *called_from, *failing_func;		
3245		{		
3246		return lm_error("ts: ts failed\n", called_from, failing_func);		
3247		}		
3248		}		
3249		}		
3250		/* void tmg_getc(ts, sprompt)		
3251		{		
3252		/* INPUT: ts = input character		
3253		/* sprompt = address of input prompt		
3254		/* OUTPUT: none		
3255		/* DESCRIPTION: Gets char value from keyboard.		
3256		{		
3257		/*		
3258		void		
3259		tmg_getc(c, prompt)		
3260		char *c,		
3261		char *prompt;		
3262		{		
3263		char reply[2];		
3264		char buffer[1024];		
3265		{		
3266		/* Get input from keyboard */		
3267		sprintf(buffer, "Enter ts : ", prompt);		
3268		#ifdef DIAGS		
3269		lm_get_input(buffer, reply, 2);		
3270		#endif DIAGS		
3271		*c = reply[0];		
3272		}		
3273		}		
3274		/*		
3275		*****		
3276		/* Debug flag routines */		
3277		*****		
3278		{		
3279		/* Allow the user in "wizard" mode to set/reset the various		
3280		/* debug flags (tmg_cal_debug) to turn on/off all the wonderful		
3281		/* messages available. This is mostly a software debugging aid, but		
3282		/* can come in handy debugging hardware, too.		
3283		{		
3284		#define WHAT(x) (tmg_cal_debug & (x) ? "ON" : "OFF")		
3285		#define FLIP(x) (tmg_cal_debug ~ (x))		
3286		long tmg_modify_edge_related_debug_flag()		
3287		{		
3288		char answer;		
3289		{		
3290		do		
3291		{		
3292		answer = 0;		
3293		lm_message("\nEDGE related debug flags are currently set as follows:\n");		
3294		lm_message("\tSingle stop (ts)\n", WHAT(1));		
3295		lm_message("\tPrint Arguments (ts)\n", WHAT(2));		
3296		lm_message("\tPrint Process Messages (ts)\n", WHAT(4));		
3297		lm_message("\tPrint Output Messages (ts)\n", WHAT(8));		
3298		lm_message("\tPrint starting points (ts)\n", WHAT(0x10));		
3299		tmg_getc(&answer, "flag to toggle, 'Q' to end");		
3300		switch(answer)		
3301		{		
3302		case 's':		
3303		case 'S': FLIP(1); break;		
3304		case 'a':		
3305		case 'A': FLIP(2); break;		
3306		case 'p':		
3307		case 'P': FLIP(4); break;		
3308		case 'o':		
3309		case 'O': FLIP(8); break;		
3310		case 't':		
3311		case 'T': FLIP(0x10); break;		
3312		default: break;		
3313		}		
3314		} while((answer != 'q') && (answer != 'Q'));		
3315		return SUCCESS;		
3316		}		
3317		long tmg_modify_sample_related_debug_flag()		
3318		{		
3319		char answer;		
3320		{		
3321		do		
3322		{		
3323		answer = 0;		
3324		lm_message("\nSAMPLE related debug flags are currently set as follows:\n");		
3325		lm_message("\tSingle stop (ts)\n", WHAT(0x100));		
3326		lm_message("\tPrint Arguments (ts)\n", WHAT(0x200));		
3327		lm_message("\tPrint Process Messages (ts)\n", WHAT(0x400));		
3328		lm_message("\tPrint Output Messages (ts)\n", WHAT(0x800));		
3329		lm_message("\tPrint starting points (ts)\n", WHAT(0x1000));		
3330		tmg_getc(&answer, "flag to toggle, 'Q' to end");		
3331		switch(answer)		
3332		{		
3333		case 's':		
3334		case 'S': FLIP(0x100); break;		
3335		case 'a':		
3336		case 'A': FLIP(0x200); break;		
3337		case 'p':		
3338		case 'P': FLIP(0x400); break;		
3339		case 'o':		
3340		case 'O': FLIP(0x800); break;		
3341		case 't':		
3342		case 'T': FLIP(0x1000); break;		
3343		default: break;		
3344		}		
3345		}		
3346		} while((answer != 'q') && (answer != 'Q'));		
3347		return SUCCESS;		
3348		}		
3349		}		
3350		long tmg_modify_delayline_related_debug_flag()		
3351		{		
3352		char answer;		
3353		{		
3354		do		
3355		{		
3356		answer = 0;		
3357		lm_message("\nDELAY LINE related debug flags are currently set as follows:\n");		
3358		lm_message("\tCalibrate delay output Messages (ts)\n", WHAT(0x10000));		
3359		lm_message("\tCompute delay process Messages (ts)\n", WHAT(0x20000));		
3360		{		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM <b>diags/tmg_cal.c</b>	DATE 5/23/89	PAGE # 29/176
			TIME 4:41:30 pm	

```

LINE # SOURCE TEXT
3361   ln_message("\tAdjust delay process messages (%s)\n", WEAT(0x40000));
3362   tmg_getc(&answer, "flag to toggle, 'Q' to end");
3363   switch(answer)
3364   {
3365       case 'C':
3366           case 'C': FLIP(0x10000); break;
3367       case 'O':
3368           case 'O': FLIP(0x20000); break;
3369       case 'A':
3370           case 'A': FLIP(0x40000); break;
3371       default: break;
3372   }
3373   while((answer != 'q') && (answer != 'Q'));
3374   return SUCCESS;
3375 }
3376
3377 long tmg_modify_misc_related_debug_flag()
3378 {
3379     char answer;
3380
3381     do
3382     {
3383         answer = 0;
3384         ln_message("\t\t\t\t\tMISCELLANEOUS related debug flags are currently set as follows:\n");
3385         ln_message("\t\t\t\t\tCalibration structure          (%s)\n", WEAT(0x1000000));
3386         ln_message("\t\t\t\t\tLine fit arguments              (%s)\n", WEAT(0x1000000));
3387         ln_message("\t\t\t\t\tLine fit outputs                (%s)\n", WEAT(0x2000000));
3388         ln_message("\t\t\t\t\tTiming() Messages               (%s)\n", WEAT(0x4000000));
3389         ln_message("\t\t\t\t\tSample width() mMessages        (%s)\n", WEAT(0x1000000));
3390         ln_message("\t\t\t\t\tComplete calib. structure Process Messages (%s)\n", WEAT(0x1000000));
3391         ln_message("\t\t\t\t\tAllow extra errors before abort  (%s)\n", WEAT(0x2000000));
3392         ln_message("\t\t\t\t\tSingle step major calibration sections (%s)\n", WEAT(0x4000000));
3393         ln_message("\t\t\t\t\tData logging messages            (%s)\n", WEAT(0x8000000));
3394         tmg_getc(&answer, "flag to toggle, 'Q' to end");
3395         switch(answer)
3396         {
3397             case 'C':
3398                 case 'C': FLIP(0x800000); break;
3399             case 'L':
3400                 case 'L': FLIP(0x1000000); break;
3401             case 'I':
3402                 case 'I': FLIP(0x2000000); break;
3403             case 'M':
3404                 case 'M': FLIP(0x4000000); break;
3405             case 'E':
3406                 case 'E': FLIP(0x8000000); break;
3407             case 'P':
3408                 case 'P': FLIP(0x10000000); break;
3409             case 'S':
3410                 case 'S': FLIP(0x20000000); break;
3411             case 'D':
3412                 case 'D': FLIP(0x40000000); break;
3413             default: break;
3414         }
3415         while((answer != 'q') && (answer != 'Q'));
3416         return SUCCESS;
3417     }
3418
3419 long tmg_modify_all_debug_flag()
3420 {
3421     char answer;
3422     int done = 0;
3423
3424     do
3425     {
3426         ln_message("Set or Reset all debug flags\n");
3427         tmg_getc(&answer, "Set, Reset, or Quit");
3428         switch(answer)
3429         {
3430             case 'S':
3431                 tmg_cal_debug |= 0xffffffff; done++; break;
3432             case 'R':
3433                 tmg_cal_debug ^= 0; done++; break;
3434             default: break;
3435         }
3436         while(!done);
3437         return SUCCESS;
3438     }
3439
3440 extern struct calib;
3441 print_calib_structure()
3442 {
3443     if(!tmg_is_calibrated())
3444     {
3445         ln_message("\t\t\t\t\t007 Calibration has not been successfully completed\n");
3446         ln_message("Interpret the calibration data accordingly\n");
3447     }
3448     tmg_print_calib_structure(&calib);
3449     #ifdef PS_TS_DEBUG
3450         print_ps_and_ts();
3451     #endif PS_TS_DEBUG
3452     return SUCCESS;
3453 }
3454
3455 #ifdef PS_TS_DEFON
3456 print_ps_and_ts()
3457 {
3458     int x, e, i;
3459
3460     for (x = 0; x < NUMBER_OF_ERAMPS; ++x) {
3461         for (e = 0; e < (NUMBER_OF_EDGES + 1); ++e) {
3462             for (i = 0; i < 7; ++i) {
3463                 ln_message("\t\t\t\t\t%d %d %d\n", x, e, pi[z][e][i], ti[z][e][i]);
3464             }
3465             ln_message("\n");
3466         }
3467     }
3468 }
3469 #endif PS_TS_DEBUG
3470
3471 }
```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89  
TIME 4:41:33 pm

PAGE #  
1/177

```

LINE # SOURCE TEXT
1  /* SCCS_ID: tmg_diag.c rev 1.2, 5/9/89 at 15:55:24 */
2
3  .....
4  tmg_diag.c
5  .....
6  Diagnostic routines for the Timing Generator
7  .....
8  .....
9
10 .....
11 #include "common.h"
12 #include "vrtx.h"
13 #include "cpu.h"
14 #include "lm_diags.h"
15 #include "tmg.h"
16 #include "tmg_def.h"
17 #include "mod_def.h"
18 #include "tmg_exts.h"
19 #include "id.h"
20 extern int tmg_cal_debug;
21
22 /*
23  * long tmg_register_test(void)
24  * Performs Register test of all read/write registers
25  *
26  * Input: none
27  * Returns: SUCCESS if no errors, else FAILURE
28  */
29 tmg_register_test()
30 {
31     int i, returncode;
32     static unsigned char misc[] = {0x10, 2, 3, 4, 5, 6, 7, 8};
33     unsigned char tmp;
34
35     if(tmg_clockoff() != SUCCESS) /* turn clock off so as not to interfere */
36         return FAILURE;
37
38     returncode = SUCCESS;
39
40     /* First check to see that all bits of all registers work */
41     /* Register 0 is weird in that it has OMRSET that clears bits 0 and 1 */
42     /* so it gets its own walk1 and walk0 test. */
43     /* Register 2 is special because it is the PLL rate register and is not */
44     /* allowed to contain some values. It can contain only values between */
45     /* 1 and 129. The walking 0 test has been modified accordingly. */
46     for(i = 0; i < 7; i++)
47     {
48         switch(i)
49         {
50             case 0:
51                 if(tmg_reg0walk1(REG(i)) != SUCCESS)
52                     returncode = FAILURE;
53                 if(tmg_reg0walk0(REG(i)) != SUCCESS)
54                     returncode = FAILURE;
55                 break;
56             case 2:
57                 if(tmg_walk1(REG(i)) != SUCCESS)
58                     returncode = FAILURE;
59                 if(tmg_reg2walk0(REG(i)) != SUCCESS)
60                     returncode = FAILURE;
61                 break;
62             default:
63                 if(tmg_walk1(REG(i)) != SUCCESS)
64                     returncode = FAILURE;
65                 if(tmg_walk0(REG(i)) != SUCCESS)
66                     returncode = FAILURE;
67                 break;
68         }
69     }
70
71     /* Next, see that all registers are independently addressable */
72     for(i = 0; i < 8; i++)
73     {
74         *REG(i) = misc[i];
75
76         for(i = 0; i < 8; i++)
77         {
78             if((tmp = *REG(i)) != misc[i])
79             {
80                 (void)lm_error("Register at %02lx: wrote %02x, read %02lx",
81                               (u_long)(REG(i)), misc[i], tmp);
82                 returncode = FAILURE;
83             }
84         }
85     }
86
87     if (tmg_reset(TRUE) != SUCCESS) /* force hard reset after this test */
88     {
89         (void)lm_warning("Reset after register test failed.\n");
90     }
91
92     return(returncode);
93 }
94
95 /*
96  * long tmg_idprom_test(void)
97  * Input: none
98  * Returns: SUCCESS or FAILURE
99  */
100 tmg_idprom_test()
101 {
102     u_char tmp;
103     if((tmp = id_checksum((u_char *)((int)&tmgptr->id_prom[0] + 3)))
104        != ID_CHECKSUM_GOOD)
105     {
106         (void)lm_error("ID Prom checksum %02X (should be %02X)\n", tmp, ID_CHECKSUM_GOOD);
107         return FAILURE;
108     }
109
110     return SUCCESS;
111 }
112
113 /*
114  * long tmg_ctc_test(void)
115  * Input: none
116  * Returns: SUCCESS or FAILURE
117  *
118  * Note: this routine relies on the clocks working.
119  * It will be hard to know if the CTC counts wrong or
120  * if the PLL generates the wrong clock frequency.
121  */
122 tmg_ctc_test()

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_diag.c	DATE 5/23/89	PAGE # 2/178
LINE #		SOURCE TEXT		
121		int returncode, i;		
122		int ctrl[3], ctr2[3];		
123		unsigned long start;		
124		int zero = 0;		
125				
126		returncode = SUCCESS;		
127		if(tmg_clockoff() != SUCCESS) /* shut clock off if on */		
128		{		
129		(void)lm_error("Cannot continue CTC Test with broken clock.\n");		
130		return(FAILURE);		
131		}		
132				
133		tmgptr->pll_rate = (256 - 213) * 2; /* freq=30Mhz*213/256 */		
134		tmgptr->pll_divisor = 255; /* just divide by 2 */		
135		tmgptr->clock_select = 0; /* select divide by 2 */		
136				
137		lm_delay(2); /* lots of lock time */		
138		if(tmg_clockon() != SUCCESS) /* turn it on */		
139		{		
140		(void)lm_error("Cannot continue CTC Test, clock does not turn on.\n");		
141		return(FAILURE);		
142		}		
143		tmgptr->ctc_intr_clearl = 0; /* make sure output is false */		
144		if(tmgptr->ctc_intr)		
145		{		
146		(void)lm_error("CTC Test: Interrupt flip flop did not clear.\n");		
147		tmgptr->ctc_register[3].value = CTRL0CH; /* control word */		
148		tmgptr->ctc_register[0].value = 1; /* very short count, lab */		
149		tmgptr->ctc_register[0].value = zero; /* mab */		
150		tmgptr->ctc_intr_clearl = 1; /* trigger timer */		
151		for(i = 0; i < 10; i++) /* wait just a little while */		
152		{		
153		if(!tmgptr->ctc_intr)		
154		{		
155		(void)lm_error("CTC Test: Interrupt failure after short count.\n");		
156		returncode = FAILURE;		
157		}		
158		}		
159				
160		tmgptr->ctc_intr_clearl = 0; /* make sure output is false */		
161		if(tmgptr->ctc_intr)		
162		{		
163		(void)lm_error("CTC Test: Interrupt flip flop did not clear.\n");		
164		returncode = FAILURE;		
165		}		
166				
167		tmgptr->ctc_register[3].value = CTRL0CH; /* control word */		
168		tmgptr->ctc_register[0].value = 0x00; /* lab of 1000 */		
169		tmgptr->ctc_register[0].value = 0x03; /* mab of 1000 */		
170				
171		tmgptr->ctc_register[3].value = CTRL1CH; /* control word */		
172		tmgptr->ctc_register[1].value = zero; /* max count 2**32 */		
173		tmgptr->ctc_register[1].value = zero;		
174				
175		tmgptr->ctc_register[3].value = CTRL2CH; /* control word */		
176		tmgptr->ctc_register[2].value = zero; /* max count 2**32 */		
177		tmgptr->ctc_register[2].value = zero;		
178				
179		tmgptr->ctc_intr_clearl = 1; /* start counter 0 */		
180				
181		start = lm_time();		
182		while(!tmgptr->ctc_intr) /* wait for done */		
183		if((lm_time() - start) > 10)		
184		{		
185		(void)lm_error("CTC Test: Timer test did not finish.\n");		
186		returncode = FAILURE;		
187		break;		
188		}		
189				
190		tmgptr->ctc_register[3].value = CTRL1LATCH; /* latch count/status */		
191				
192		/* counter 0 loaded with count of 1000 */		
193		/* counters 1,2 are loaded with 0, and count down */		
194		/* freq of counter 0 clock is (213/256)*30 Mhz */		
195		/* freq of counters 1,2 is 30 Mhz / 4 */		
196		/* should count 1000*256/4/213/2 = 2403.7 = 963h clocks */		
197		/* this results in reading 10000h - 963h = f63dh from */		
198		/* counter 1 and nothing from counter 2 */		
199		/* Now, it is easy to get +/- one clock accuracy since the */		
200		/* two clock frequencies are almost relatively prime, and */		
201		/* with a little PLL error and the fact that 7 clocks is */		
202		/* close to a whole clock, it is reasonable to expect that */		
203		/* we could measure 963h +2,-1 clocks, or read f63dh thru */		
204		/* f69eh. If we read outside this range, it is an error. */		
205				
206				
207		for(i = 0; i < 3; i++)		
208		ctrl[i] = tmgptr->ctc_register[i].value & 0x0ff;		
209		if(!(ctrl[i] & 0x00)) /* null flag not set */		
210		{		
211		if((ctrl[i] < 0x9a)    (ctrl[i] > 0x9e)) /* lab first */		
212		{		
213		(void)lm_error("CTC Test: Count test, register 1 lab = 002x, \n		
214		expected from 9a thru 9e.\n", ctrl[i]);		
215		returncode = FAILURE;		
216		}		
217				
218		if(ctrl[i] & 0x0ff) /* mab next */		
219		{		
220		(void)lm_error("CTC Test: Count test, register 1 mab = 002x, expected f6.\n", ctrl[i]);		
221		returncode = FAILURE;		
222		}		
223		else		
224		{		
225		(void)lm_error("CTC Test: Counter 1 Null flag set.\n");		
226		returncode = FAILURE;		
227		}		
228				
229		for(i = 0; i < 3; i++)		
230		ctrl2[i] = tmgptr->ctc_register[i].value & 0x0ff;		
231		if(!(ctrl2[i] & 0x00)) /* null flag not set */		
232		{		
233		(void)lm_error("CTC Test: Count test, counter 2 null flag not set.\n");		
234		returncode = FAILURE;		
235		}		
236				
237		return(returncode);		
238				
239				
240				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:33 pm	3/179

```

LINE #
241
242 /* int tmg_plltime(timeptr)
243 /* Measure lock time of PLL
244 /* Input: pointer to lock time
245 /* Output: time is assigned, function returns SUCCESS or FAILURE
246
247 /* This routine uses the 8254 in a strange way
248 /* Counter 1 is used to count the time since it's clock
249 /* is FREQ32 (3.75 MHz), but the gate from Counter 0 output
250 /* must be high.
251 /* 1) First a fairly slow clock is selected
252 /* 2) Counter 0 is set for mode 1 with a long count
253 /* 3) As soon as it's count is started, the clock should be
254 /* turned off, leaving it's output active (gate for Counter 1)
255 /* 4) Load Counter 1
256 /* 5) Monkey with PLL
257 /* 6) When PLL is locked, check count in Counter 1, convert
258 /* to time
259 */
260 int tmg_plltime(timeptr)
261 unsigned long *timeptr;
262 {
263     long i, usecs, maxtries, slowuptime, slowdowntime;
264     unsigned long value, start;
265     int ctrl[3], debounced;
266     int successes, failures, successsthistime, failedthistime;
267     int upreturcode, downreturcode;
268     int zero = 0;
269
270     upreturcode = downreturcode = SUCCESS;
271
272     if(tmg_clockoff() != SUCCESS) /* make sure clock is off */
273     {
274         (void)lm_error("PLL Lock Time: cannot turn clock off(1)\n");
275         return(FAILURE);
276     }
277
278     tmgptr->pll_rate = (256 - 128) * 1; /* set to 30 MHz */
279     tmgptr->pll_divisor = (256 - 15) * 1; /* choose div 2 * 15 */
280     tmgptr->clock_select = 1; /* sel div 2K */
281
282     lm_delay(2); /* wait lots of time for lock */
283
284     if(!tmgptr->pll_locked)
285     {
286         (void)lm_error("PLL Lock Time: Lock indicator not functional\n");
287         return(FAILURE);
288     }
289
290     if(tmg_clockon() != SUCCESS) /* make sure clock is on */
291     {
292         (void)lm_error("PLL Lock Time: cannot turn clock on\n");
293         return(FAILURE);
294     }
295
296     tmgptr->ctc_intr_clearl = 0; /* GATE0 low */
297     tmgptr->ctc_register[3].value = 0x30; /* setup mode 0 */
298     tmgptr->ctc_register[0].value = zero; /* lab for max count */
299     tmgptr->ctc_register[0].value = zero; /* mab */
300
301     /* since GATE0 has no effect on OUT0, OUT0 should be low */
302     /* and furthermore, Counter0 should not be counting */
303
304     /* at this point, merely verify that Counter0 output is low */
305     maxtries = 10000;
306     while(1)
307     {
308         tmgptr->ctc_register[3].value = LATCH_OUT0;
309         if((tmgptr->ctc_register[0].value & 0x80))
310             break;
311         if(--maxtries == 0)
312         {
313             (void)lm_error("PLL Lock Time: Counter0 output was not low\n");
314             return FAILURE;
315         }
316     }
317
318     /* turn the clock off to keep the PACs & PLLs happy */
319     if(tmg_clockoff() != SUCCESS) /* make sure clock is off */
320     {
321         (void)lm_error("PLL Lock Time: cannot turn clock off(2)\n");
322         return(FAILURE);
323     }
324
325     /* measure time to lock from 30 -> 60 MHz */
326     successes = failures = 0;
327     slowuptime = 0;
328     do
329     {
330         tmgptr->pll_rate = 129; /* set to 30 MHz */
331         lm_delay(2); /* wait for lock */
332         maxtries = 1000; /* cpu speed dependent */
333         successsthistime = failedthistime = FALSE;
334         tmgptr->ctc_register[3].value = 0x070; /* setup mode 0 */
335         tmgptr->ctc_register[1].value = zero; /* counter now running */
336         tmgptr->ctc_register[1].value = zero;
337
338         CPU_DISABLE_INTERRUPTS;
339         tmgptr->pll_rate = 1; /* (256 - 256) * 1 */
340         /* slow to 60 MHz */
341
342         while(tmgptr->pll_locked == 1)
343         {
344             if(--maxtries == 0)
345             {
346                 ++failures;
347                 failedthistime = TRUE;
348                 break;
349             }
350
351             if(failures >= 100)
352             {
353                 CPU_ENABLE_INTERRUPTS;
354                 break; /* break out of loop */
355             }
356             if(failedthistime == TRUE)
357             {
358                 CPU_ENABLE_INTERRUPTS;
359                 continue; /* go try again */
360             }
361             else
362             {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:33 pm	4/180

```

LINE #      SOURCE TEXT
361      ++successes; /* must have been a success */
362      successsthistime = TRUE;
363      }
364
365      debounced = 0;
366      maxtries = 1000;
367      while(!debounced) /* wait for lock and debounce, too */
368      {
369          for(i = 0; i < 50; i++) /* look for 50 samples in row */
370          {
371              if(!(--maxtries))
372              {
373                  CPU_ENABLE_INTERRUPTS;
374                  (void)lm_error("PLL Lock Time: Timeout waiting for lock true(1)\n");
375                  return FAILURE;
376              }
377              if(!tmgptr->pll_locked)
378                  break;
379              if(i == 0) /* get the time of first lock */
380              {
381                  tmgptr->ctc_register[3].value = LATCH_CNTR_1;
382                  for(i = 0; i < 3; i++)
383                      cntl[i] = tmgptr->ctc_register[i].value & 0x0ff;
384                  if(cntl[0] & 0x40) /* if null flag set */
385                      value = 01;
386                  else
387                  {
388                      value = cntl[1] + (cntl[2] << 8);
389                      if(value == 01)
390                          value = 11;
391                      else
392                          value = 0x100011 - value;
393                  }
394              }
395              if(i == 50)
396                  debounced = 1;
397          }
398          CPU_ENABLE_INTERRUPTS;
399          if(successsthistime == TRUE)
400              slowuptime += value;
401          while(successes < 10);
402      }
403      if(successes == 0)
404      {
405          upreturacode = FAILURE;
406      }
407      else
408      {
409          slowuptime = slowuptime * 2567 / (10000 * successes);
410          lm_message("slow up time = %d usons\n", slowuptime);
411      }
412
413      /* now measure time to lock from 50 -> 30 MHz */
414      successes = failures = 0;
415      slowdowntime = 0;
416      do
417      {
418          tmgptr->pll_rate = 1; /* set to 50 MHz */
419          lm_delay(2); /* wait for lock */
420          maxtries = 1000; /* cpu speed dependent */
421          successsthistime = failedthistime = FALSE;
422          tmgptr->ctc_register[3].value = 0x070; /* setup mode 0 */
423          tmgptr->ctc_register[1].value = zero; /* counter now running */
424          tmgptr->ctc_register[1].value = zero;
425
426          CPU_DISABLE_INTERRUPTS;
427          tmgptr->pll_rate = 129; /* (256 - 128) + 1 */
428          /* slow to 30 MHz */
429          while(tmgptr->pll_locked == 1)
430          {
431              if(!--maxtries)
432              {
433                  ++failures;
434                  failedthistime = TRUE;
435                  break;
436              }
437          }
438          if(failures >= 100)
439          {
440              CPU_ENABLE_INTERRUPTS;
441              break; /* break out of loop */
442          }
443          if(failedthistime == TRUE)
444          {
445              CPU_ENABLE_INTERRUPTS;
446              continue; /* go try again */
447          }
448          else
449          {
450              ++successes; /* must have been a success */
451              successsthistime = TRUE;
452          }
453      }
454      debounced = 0;
455      maxtries = 1000;
456      while(!debounced) /* wait for lock and debounce, too */
457      {
458          for(i = 0; i < 50; i++) /* look for 50 samples in row */
459          {
460              if(!(--maxtries))
461              {
462                  CPU_ENABLE_INTERRUPTS;
463                  (void)lm_error("PLL Lock Time: Timeout waiting for lock true(2)\n");
464                  return FAILURE;
465              }
466              if(!tmgptr->pll_locked)
467                  break;
468              if(i == 0) /* get the time of first lock */
469              {
470                  tmgptr->ctc_register[3].value = LATCH_CNTR_1;
471                  for(i = 0; i < 3; i++)
472                      cntl[i] = tmgptr->ctc_register[i].value & 0x0ff;
473                  if(cntl[0] & 0x40) /* if null flag set */
474                      value = 01;
475                  else
476                  {
477                      value = cntl[1] + (cntl[2] << 8);
478                      if(value == 01)
479                          value = 11;
480

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE	5/23/89	PAGE #
TIME	4:41:33 pm	5/181

```

LINE # SOURCE TEXT
481     else
482     value = success1 - value;
483     }
484     }
485     if(i == 50)
486     {
487         debounce = 1;
488     }
489     CPU_ENABLE_INTERRUPTS;
490     if(success1time == TRUE)
491     {
492         slowdowntime = value;
493         while(successes < 10);
494         if(successes == 0)
495         {
496             dowarreturncode = FAILURE;
497         }
498     }
499     else
500     {
501         slowdowntime = slowdowntime + 2667 / (10000 * successes);
502         /* message("slow down time = %d usecs\n", slowdowntime); */
503     }
504     usecs = (slowuptime > slowdowntime) ? slowuptime : slowdowntime;
505     *timeptr = usecs;
506     if((upreturncode == SUCCESS) || (dowarreturncode == SUCCESS))
507     {
508         return SUCCESS;
509     }
510     else
511     {
512         return FAILURE;
513     }
514 }
515
516 // long tmg_pll_lock(void)
517 //
518 // This function checks PLL lock time against limits. It
519 // calls tmg_plltime() to measure the time.
520 //
521 // Inputs: none
522 // Outputs: function returns SUCCESS or FAILURE
523 //
524 tmg_pll_lock()
525 {
526     u_long locktime;
527     if(tmg_plltime(&locktime) != SUCCESS)
528     {
529         (void)lm_error("PLL Lock Time: Error measuring lock time.\n");
530         return(FAILURE);
531     }
532     (void)lm_message("PLL Lock Time is %ld microseconds.\n", locktime);
533     if(locktime > 1500)
534     {
535         lm_error("PLL Lock Time (%d usec) exceeds 1500 usec bounds.\n");
536         return FAILURE;
537     }
538     else
539     {
540         return(SUCCESS);
541     }
542 }
543
544 // long tmg_pll_rate(void)
545 // This routine checks the PLL frequencies for various
546 // values of N to verify all aspects of divide-by-N counter
547 //
548 // Counter load values will be: 1 (N=256)
549 //                               128 (N=128)
550 //                               55a (N=172)
551 //                               2ah (N=42)
552 //
553 // Input: nothing
554 // Return: SUCCESS or FAILURE
555 //
556 // Test failures are those where tmg_checkrate() returns
557 // discrepancy of more than 1 clock.
558 //
559 // PLL freq: fo = 30 MHz / 256
560 // TTL000V CLK freq: TTL000V_CLK / 8
561 // TTL000V CLK freq: either fo/2, fo/2K, or fo/4K
562 // TTL000V12 freq: 30 MHz / 8 (reference to 254)
563 //
564 // The 254 counts the number of reference clocks (TTL000V12)
565 // that occur within 1000 unknown clocks (TTL000V). This results
566 // in a measurement that is accurate to at least 1 part in 1000
567 // for the unknown frequency.
568 //
569 // The time for 1000 clocks of unknown frequency is found to be
570 // t = 1000 / ((N * (30 MHz / 256)) / 16) / X, X = 2, 2K, or 4K
571 //
572 // The number of reference clocks that expire in an interval t is
573 // flocks = t * (30 MHz / 8)
574 //         = 1000 / ((N / 256) / X) * X = 2, 2K, or 4K
575 //
576 // Even if the PLL were perfect, there could be +/- 1 clock in our
577 // measurement. Some of the calculated number of clocks are non-integers
578 // that with a little error on the PLL, could result in another clock
579 // being measured. Also, depending on how many of the PLL reference
580 // periods (117 KHz) occur within the measurement interval, can get
581 // more error. It is reasonable to expect 0.1% error over the short
582 // haul.
583 //
584 tmg_pll_rate()
585 {
586     int i;
587     int returncode;
588     int noclcks;
589     /* N = 256 => fo = 30 MHz, should count 2000 clocks */
590     /* N = 128 => fo = 30 MHz, should count 4000 clocks */
591     /* N = 172 => fo = 40.3125 MHz, should count 2976 clocks */
592     /* N = 215 => fo = 50.390625 MHz, should count 2381 clocks */
593     static int n[] = { 256, 128, 172, 215 };
594     static long clocks[] = { 2000L, 4000L, 2976L, 2381L };
595     static int delta[] = { 3, 5, 5, 4 };
596     static char error_format[]
597     = "%s Divide by %d failure.\nExpected %d clocks, measured %d clocks\n";
598     returncode = SUCCESS;
599     for (i = 0; i < 4; ++i) {
600         if((tmg_checkrate(n[i], &noclcks) != SUCCESS))

```



1217

5,353,243

1218

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_diag.c	DATE 5/23/89	PAGE # 6/182
LINE #	SOURCE TEXT			
601	returncode = FAILURE;			
602	if (lm_error("PLL Div by N: tmg_checkrate() returned error.\n");			
603	!= SUCCESS) return FAILURE;			
604	else			
605	{			
606	if(abs(soclocks) > Delta[i]) /* 10 + 10000 */			
607	{			
608	returncode = FAILURE;			
609	if (lm_error(error_format, "PLL Div by N:",			
610	s[i], clocks[i], clocks[i] + soclocks) != SUCCESS)			
611	return FAILURE;			
612	}			
613	}			
614	return(returncode);			
615	}			
616	/*			
617	loop tmg_pll_div(void)			
618	/* this test checks divide by 2K counter and last div 4K flip flop			
619	/* All tests run using T= 60 MHz			
620	/* Input: nothing			
621	/* Returns: SUCCESS or FAILURE			
622	*/			
623	tmg_pll_div()			
624	{			
625	int returncode;			
626	int error;			
627	returncode = SUCCESS;			
628	/* 2K, K=2, fclocks = 1000 / ((K/256)/2K) N = 256			
629	if(tmg_checkrate(256,4,40001,error) != SUCCESS)			
630	return(FAILURE);			
631	else			
632	if(abs(error) > 5) /* 10 + 10000 */			
633	{			
634	(void)lm_error("PLL Div by K: Divide by 2 failure.\n");			
635	returncode = FAILURE;			
636	}			
637	/* 2K, K = 256			
638	if(tmg_checkrate(256,512,5120001,error) != SUCCESS)			
639	return(FAILURE);			
640	else			
641	if(abs(error) > 5) /* can't believe it could be that high */			
642	{			
643	(void)lm_error("PLL Div by K: Divide by 256 failure.\n");			
644	returncode = FAILURE;			
645	}			
646	/* 2K, K = 172 (pll_divisor = 8x35)			
647	if(tmg_checkrate(256,344,3440001,error) != SUCCESS)			
648	return(FAILURE);			
649	else			
650	if(abs(error) > 5) /* same as above */			
651	{			
652	(void)lm_error("PLL Div by K: Divide by 172 failure.\n");			
653	returncode = FAILURE;			
654	}			
655	/* 2K, K = 87 (pll_divisor = 8x88)			
656	if(tmg_checkrate(256,174,1740001,error) != SUCCESS)			
657	return(FAILURE);			
658	else			
659	if(abs(error) > 5) /* same as above */			
660	{			
661	(void)lm_error("PLL Div by K: Divide by 87 failure.\n");			
662	returncode = FAILURE;			
663	}			
664	return(returncode);			
665	/*			
666	tmg_slow_detect(void)			
667	/*			
668	checks slow clock detector			
669	Inputs: none			
670	Returns: SUCCESS or FAILURE			
671	*/			
672	tmg_slow_detect()			
673	{			
674	int returncode;			
675	u_long start;			
676	returncode = SUCCESS;			
677	if(tmg_clockoff() != SUCCESS)			
678	{			
679	(void)lm_error("Slow Clock Detect: Could not turn clock off.\n");			
680	return(FAILURE);			
681	}			
682	tmgptr->slow_clock_clear1 = 0;			
683	if(tmgptr->slow_clock)			
684	{			
685	(void)lm_error("Slow Clock Detect: Slow status bit stuck true.\n");			
686	tmgptr->slow_clock_clear1 = 0;			
687	return(FAILURE);			
688	}			
689	tmgptr->pll_rate = 1; /* PLL set to 60 MHz */			
690	tmgptr->pll_divisor = 33; /* F = 224 (134 KHz) */			
691	tmgptr->clock_select = 1; /* Select divide by 2K */			
692	lm_delay(10); /* wait for lock */			
693	if(tmg_clockon() != SUCCESS)			
694	{			
695	(void)lm_error("Slow Clock Detect: cannot turn clock on.\n");			
696	return(FAILURE);			
697	}			
698	start = lm_time();			
699	do			
700	{			
701	if((lm_time() - start) > TIMEOUT)			
702	{			
703	(void)lm_error("Slow Clock Detect: timeout trying to initialize.\n");			
704	return FAILURE;			
705	}			
706	tmgptr->slow_clock_clear1 = 0; /* clear it */			
707	tmgptr->slow_clock_clear1 = 1; /* unclear it */			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:33 pm 7/183

```

LINE # SOURCE TEXT
721 } while(tmgptr->slow_clock), /* it set, repeat */
722 /* now the flip flop is really cleared, lets see if it sets */
723 lm_delay(10); /* detector should not fire */
724 /* is about 8 usecs */
725 if(tmgptr->slow_clock)
726 {
727     (void)lm_error("Slow Clock Detect: detector fired at 134 KHz.\n");
728     returncode = FAILURE;
729 }
730 if(tmg_clockoff() != SUCCESS)
731 {
732     (void)lm_error("Slow Clock Detect: cannot turn clock off.\n");
733     return(FAILURE);
734 }
735 tmgptr->slow_clock_clear1 = 0;
736 tmgptr->pll_divisor = 1; /* K = 256 (117 KHz) */
737 if(tmg_clockon() != SUCCESS)
738 {
739     (void)lm_error("Slow Clock Detect: cannot turn clock on.\n");
740     return(FAILURE);
741 }
742 lm_delay(1); /* wait for freq to stabilize */
743 tmgptr->slow_clock_clear1 = 1;
744 lm_delay(10); /* detector should fire */
745 /* is about 8 usecs */
746 if(!tmgptr->slow_clock) /* should be set */
747 {
748     (void)lm_error("Slow Clock Detect: detector did not fire at 117 KHz.\n");
749     returncode = FAILURE;
750 }
751 if(tmg_clockoff() != SUCCESS)
752 {
753     (void)lm_error("Slow Clock Detect: cannot turn clock off.\n");
754     return(FAILURE);
755 }
756 tmgptr->pll_divisor = 33; /* K = 324 (134 KHz) */
757 lm_delay(1);
758 if(tmg_clockon() != SUCCESS)
759 {
760     (void)lm_error("Slow Clock Detect: cannot turn clock on.\n");
761     return(FAILURE);
762 }
763 if(!tmgptr->slow_clock) /* should still be set */
764 {
765     (void)lm_error("Slow Clock Detect: output cleared without command.\n");
766     returncode = FAILURE;
767 }
768 tmgptr->slow_clock_clear1 = 0;
769 return(returncode);
770 }
771
772 /*
773  * int tmg_freq_ext0(void)
774  * This function measures the frequency of External Clock 0.
775  * Inputs: none
776  * Outputs: function returns SUCCESS or FAILURE
777  */
778 int tmg_freq_ext0()
779 {
780     unsigned long hertz, noclocks;
781     if(tmg_clockoff() != SUCCESS)
782     {
783         (void)lm_error("Measure Ext0 Frequency: cannot shut clock off.\n");
784         return(FAILURE);
785     }
786     tmgptr->clock_select = 3;
787     if(tmg_measure_freq(&noclocks) != SUCCESS)
788     {
789         (void)lm_error("Measure Ext0 Frequency: FAILURE!\n");
790         return(FAILURE);
791     }
792     hertz = (60ll / (double)noclocks);
793     (void)lm_banner("External Clock 0 frequency is %ld Hz.\n", hertz);
794     return(SUCCESS);
795 }
796
797 /*
798  * int tmg_freq_ext1(void)
799  * This function measures the frequency of External Clock 1.
800  * Inputs: none
801  * Outputs: function returns SUCCESS or FAILURE
802  */
803 int tmg_freq_ext1()
804 {
805     unsigned long hertz, noclocks;
806     if(tmg_clockoff() != SUCCESS)
807     {
808         (void)lm_error("Measure Ext1 Frequency: cannot shut clock off.\n");
809         return(FAILURE);
810     }
811     tmgptr->clock_select = 4;
812     if(tmg_measure_freq(&noclocks) != SUCCESS)
813     {
814         (void)lm_error("Measure Ext1 Frequency: FAILURE!\n");
815         return(FAILURE);
816     }
817     hertz = (60ll / (double)noclocks);
818     (void)lm_banner("External Clock 1 frequency is %ld Hz.\n", hertz);
819     return(SUCCESS);
820 }
821
822 /*
823  * long tmg_interrupts(void)
824  * This function tests Timing Generator Interrupt capability.
825  * Inputs: none
826  * Outputs: function returns SUCCESS or FAILURE
827  */
828 long tmg_interrupts()
829 {
830     long returncode;
831     unsigned long start;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89  
TIME 4:41:33 pm

PAGE #  
8/184

```

LINE #          SOURCE TEXT
841  int zero = 0;
842
843  returncode = SUCCESS;
844
845  if(tmg_reset(FALSE) != SUCCESS)
846  {
847      (void)lm_error("Interrupt Test: cannot reset.\n");
848      return(FAILURE);
849  }
850
851  if(tmg_clockoff() != SUCCESS)
852  {
853      (void)lm_error("Interrupt Test: cannot turn clock off.\n");
854      if(tmg_clockon() != SUCCESS)
855      {
856          (void)lm_error("Interrupt Test: cannot turn clock on.\n");
857          return(FAILURE);
858      }
859  }
860  /* first check that no interrupts exist */
861  if(TMG_INT)
862  {
863      (void)lm_error("Interrupt Test: interrupt active after reset, check CPU.\n");
864      return(FAILURE);
865  }
866  /* check CTC Interrupt Capability */
867  (void)lm_message("Checking CTC Interrupt\n");
868  if(tmgptr->ctc_intr)
869  {
870      (void)lm_error("Interrupt Test: CTC Interrupt active after reset.\n");
871      returncode = FAILURE;
872  }
873
874  tmgptr->ctc_register[3].value = CTR0CN; /* initialize ctr.0 */
875  tmgptr->ctc_intr_enable = 1;
876  tmgptr->ctc_register[0].value = 1; /* lab of very short count */
877  tmgptr->ctc_register[0].value = zero; /* sub of very short count */
878  tmgptr->ctc_intr_clear = 1;
879  start = lm_time(); /* start timer */
880  while(!TMG_INT)
881  {
882      if((lm_time() - start) > 10)
883      {
884          (void)lm_error("Interrupt Test: CTC failed to interrupt.\n");
885          returncode = FAILURE;
886          break;
887      }
888      if(!tmgptr->ctc_intr)
889      {
890          (void)lm_error("Interrupt Test: expected CTC interrupt, received other.\n");
891          returncode = FAILURE;
892      }
893      else
894      {
895          (void)lm_message("Interrupt Test: CTC interrupt received OK.\n");
896          tmgptr->ctc_intr_clear = 0; /* clear output */
897          tmgptr->ctc_intr_enable = 0; /* disable interrupt */
898      }
899  }
900  /* now check EOP Interrupt Capability */
901  (void)lm_message("Checking End-of-Play Interrupt\n");
902  tmg_ptr->test_mode(1);
903  tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */
904  tmgptr->start_pattern_play = 1; /* start the PLAY */
905  start = lm_time();
906  while(!TMG_INT)
907  {
908      if((lm_time() - start) > 10)
909      {
910          (void)lm_error("Interrupt Test: EOP failed to interrupt.\n");
911          returncode = FAILURE;
912          break;
913      }
914      if(TMG_INT)
915      {
916          (void)lm_message("Interrupt Test: EOP interrupt received OK.\n");
917          tmgptr->pattern_intr_enable = 0;
918          if(TMG_INT)
919          {
920              (void)lm_error("Interrupt Test: EOP interrupt did not clear.\n");
921              returncode = FAILURE;
922          }
923      }
924  }
925  /* finally check Error Interrupt Capability */
926  (void)lm_message("Checking Backplane Error Interrupt\n");
927  if(tmg_clockoff() != SUCCESS)
928  {
929      (void)lm_warning("Interrupt Test: cannot turn clock off.\n");
930      if(tmg_clockon() != SUCCESS)
931      {
932          (void)lm_warning("Interrupt Test: cannot turn clock on.\n");
933          tmgptr->base_intr_enable = 1;
934          if(TMG_INT)
935          {
936              (void)lm_warning("Interrupt Test: backplane is driving ERROR.\n");
937          }
938          else
939          {
940              tmgptr->backplane_error = 1;
941              start = lm_time();
942              while(!TMG_INT)
943              {
944                  if((lm_time() - start) > 10)
945                  {
946                      (void)lm_error("Interrupt Test: Unable to cause BP Error Interrupt.\n");
947                      returncode = FAILURE;
948                      break;
949                  }
950              }
951          }
952          if(TMG_INT)
953          {
954              (void)lm_message("Interrupt Test: Backplane Error Interrupt received OK.\n");
955              tmgptr->backplane_error = 0;
956              tmgptr->base_intr_enable = 0;
957              if(TMG_INT)
958              {
959                  (void)lm_error("Interrupt Test: Backplane interrupt did not clear.\n");
960                  returncode = FAILURE;
961              }
962          }
963      }
964  }
965  return(returncode);
966
967  long tmg_loop(void)
968  {
969      /* This function is used for running continuous patterns. Actual
970       * looping is handled by menu code.
971       * Inputs: none
972  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:33 pm

9/185

```

LINE # SOURCE TEXT
951 /* Outputs: function returns SUCCESS or FAILURE
952 */
953 tmg_loop()
954 {
955     /* Make sure clock is off */
956     if (tmg_clockoff() != SUCCESS)
957         return(FAILURE);
958     if (tmg_soffclear() != SUCCESS)
959         (void)lm_warning("Loop: Cal flip flops did not clear.\n");
960     /* then turn it on */
961     if (tmg_clockon() != SUCCESS)
962         return(FAILURE);
963     if (tmg_play(TIMEOUT) != SUCCESS)
964         return(FAILURE);
965     return(SUCCESS);
966 }
967
968 /*
969  * int tmg_select_mode(void)
970  *
971  * This routine is a utility to allow setting of the desired
972  * test and sample modes from the diagnostic menus.
973  *
974  * Input: none
975  * Output: Always returns SUCCESS
976  */
977 tmg_select_mode()
978 {
979     u_long answer;
980     (void)lm_message("Select test mode:\n");
981     (void)lm_message("\t0) Normal Mode\n");
982     (void)lm_message("\t1) Test Mode 1 (1 edge ramp, 1 sample ramp) (default)\n");
983     (void)lm_message("\t2) Test Mode 2 (2 edge ramps, 1 sample ramp)\n");
984     (void)lm_message("\t3) Test Mode 3 (Continuous edge ramps)\n");
985     answer = 1; /* test mode 1 */
986     diag_get_ulong(&answer, "Choice", 01, 31);
987     tmgptr->test_mode = answer;
988     (void)lm_message("\tSelect sample mode:\n");
989     (void)lm_message("\t0) Early sample\n");
990     (void)lm_message("\t1) Edge triggered sample (default)\n");
991     answer = 1;
992     diag_get_ulong(&answer, "Choice", 01, 11);
993     tmgptr->sample_mode = answer;
994     return SUCCESS;
995 }
996
997 /*
998  * int tmg_select_ramps(void)
999  *
1000  * Choose ramps from diagnostic menus.
1001  *
1002  * Input: none
1003  * Output: Always returns SUCCESS
1004  */
1005 tmg_select_ramps()
1006 {
1007     u_long answer;
1008     (void)lm_message("Select ramps:\n");
1009     answer = 0;
1010     diag_get_ulong(&answer, "edge ramp", 01, 31);
1011     tmgptr->edge_delay_range = answer;
1012     answer = 0;
1013     diag_get_ulong(&answer, "sample ramp", 01, 31);
1014     tmgptr->sample_delay_range = answer;
1015     return SUCCESS;
1016 }
1017
1018 /*
1019  * int tmg_select_thresholds(void)
1020  *
1021  * This routine allows setting edge/sample thresholds from
1022  * the diagnostic menus.
1023  *
1024  * Input: none
1025  * Output: always returns SUCCESS
1026  */
1027 tmg_select_thresholds()
1028 {
1029     u_long answer;
1030     char buffer[32];
1031     int i;
1032     (void)lm_message("Select thresholds:\n");
1033     for(i = 0; i < 6; i++)
1034     {
1035         answer = 10;
1036         sprintf(buffer, "edge[%d] threshold", i);
1037         diag_get_ulong(&answer, buffer, 01, 2551);
1038         tmgptr->edge_delay[i].delay = answer;
1039     }
1040     answer = 10;
1041     diag_get_ulong(&answer, "sample trigger threshold", 01, 2551);
1042     tmgptr->sample_trigger_threshold = answer;
1043     answer = 10;
1044     diag_get_ulong(&answer, "sample threshold", 01, 255);
1045     tmgptr->sample_delay = answer;
1046     return SUCCESS;
1047 }
1048
1049 /*
1050  * int tmg_select_slot_count(void)
1051  *
1052  * This utility allows setting slot count from diagnostic menus.
1053  *
1054  * Input: none
1055  * Output: always returns SUCCESS
1056  */
1057 tmg_select_slot_count()
1058 {
1059     u_long answer;
1060     answer = 1;
1061     diag_get_ulong(&answer, "slot count", 11, 81);
1062     tmg_set_slot_count(answer);
1063     return SUCCESS;
1064 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:33 pm 10/186

```

LINE # SOURCE TEXT
1081 }
1082
1083
1084 /*
1085  * int tmg_edge_jitter(void)
1086  *
1087  * This utility sets up all edges at maximum threshold and allows the
1088  * operator to step through the four ramps. This makes taking an
1089  * oscilloscope measurement of the jitter as painless as possible.
1090  *
1091  * Inputs: none
1092  * Outputs: always returns SUCCESS
1093  */
1094 tmg_edge_jitter()
1095 {
1096     int ramp, edge, period;
1097     static int jitter[] = {500, 1000, 2000, 10000, 50000};
1098     lm_message("Edge Jitter Measurement\n");
1099     lm_message("Trigger the oscilloscope on rising edge of U130-15, (CECLISE_CHARGE).\n");
1100     lm_message("Jitter is the maximum uncertainty in position of the edge rising edge.\n");
1101     tmg_set_slot_count(2);
1102     tmg_set_test_mode(1);
1103     for(edge = 0; edge < 6; edge++)
1104     {
1105         tmgptr->edge_delay[edge].delay = 250;
1106         for(ramp = 0; ramp < 4; ramp++)
1107         {
1108             tmgptr->edge_delay_range = ramp;
1109             period = calib.EdgeMaxDelay[ramp] * 1.5;
1110             tmg_set_period(ramp, tperiod);
1111             lm_message("Selected edge ramp is %d\n", ramp);
1112             lm_message("Verify jitter to be less than %d ps for all 6 edges.\n",
1113                 jitter[ramp]);
1114             tmg_play_til_key();
1115         }
1116     }
1117     return SUCCESS;
1118 }
1119
1120 /*
1121  * int tmg_sample_jitter(void)
1122  *
1123  * This utility sets up the sample at maximum threshold and allows the
1124  * operator to step through the four ramps in both early and edge 7
1125  * triggered modes. This makes taking an
1126  * oscilloscope measurement of the jitter as painless as possible.
1127  *
1128  * Inputs: none
1129  * Outputs: always returns SUCCESS
1130  */
1131 tmg_sample_jitter()
1132 {
1133     int ramp, edge, period;
1134     static int jitter[] = {500, 1000, 2000, 4000, 500, 2000, 10000, 50000};
1135     lm_message("Sample Jitter Measurement\n");
1136     lm_message("Trigger the oscilloscope on rising edge of U130-15, (CECLISE_CHARGE).\n");
1137     lm_message("Jitter is the maximum uncertainty in position of the sample rising edge.\n");
1138     lm_message("Early trigger mode:\n");
1139     tmg_set_slot_count(2);
1140     tmg_set_test_mode(1);
1141     tmgptr->sample_delay = 250;
1142     tmgptr->sample_mode = 0;
1143     period = 150000;
1144     for(ramp = 0; ramp < 4; ramp++)
1145     {
1146         tmgptr->sample_delay_range = ramp;
1147         tmg_set_period(ramp, tperiod);
1148         lm_message("Selected sample ramp is %d\n", ramp);
1149         lm_message("Verify jitter to be less than %d ps.\n",
1150             jitter[ramp]);
1151         tmg_play_til_key();
1152     }
1153     lm_message("Edge 7 triggered mode:\n");
1154     tmg_set_slot_count(2);
1155     tmg_set_test_mode(1);
1156     tmgptr->sample_trigger_threshold = 250;
1157     tmgptr->sample_mode = 1;
1158     tmgptr->sample_delay_range = 0;
1159     tmgptr->sample_delay = 10;
1160     for(ramp = 0; ramp < 4; ramp++)
1161     {
1162         tmgptr->edge_delay_range = ramp;
1163         period = calib.EdgeMaxDelay[ramp] * 1.5;
1164         tmg_set_period(ramp, tperiod);
1165         lm_message("Selected edge ramp is %d\n", ramp);
1166         lm_message("Verify jitter to be less than %d ps.\n",
1167             jitter[ramp * 4]);
1168         tmg_play_til_key();
1169     }
1170     return SUCCESS;
1171 }
1172
1173 /*
1174  * int tmg_edge_alignment(void)
1175  *
1176  * This test sets up all edges to its best approximation of 50 ns
1177  * delay. The clock period is set to 100 ns. This allows the operator
1178  * to confirm edge alignment with an oscilloscope.
1179  */
1180 tmg_edge_alignment()
1181 {
1182     int edge;
1183     int period;
1184     period = 100000; /* 100000 ps */
1185     lm_message("Edge alignment test:\n");
1186     if(!calib.CalCompleted)
1187     {
1188         lm_message("Please wait while calibration is performed...\n");
1189         tmg_calibrate();
1190         if(!calib.CalCompleted)
1191         {
1192             lm_message("This test requires that calibration be successfully completed.\n");
1193             return SUCCESS;
1194         }
1195         else lm_message("\n\n");
1196     }
1197     lm_message("Target edge position is 50 ns with a clock period of 100 ns\n");
1198     lm_message("A convenient trigger is U130-15 (CECLISE_CHARGE).\n");
1199     tmg_set_slot_count(1);
1200     tmg_set_test_mode(1);

```

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM diags/tmg_diag.c	DATE	5/23/89	PAGE #
		TIME	4:41:33 pm	11/187

LINE #	SOURCE TEXT
1201	tmgptr->edge_delay_range = 0;
1202	tmgptr->sample_delay_range = 0;
1203	for(edge = 0; edge < 6; edge++)
1204	tmgptr->edge_delay[edge].delay = (500001 - (long)calib.EdgeOffset[0][edge])
1205	/ (long)calib.EdgeSlope[0][edge];
1206	tmg_set_period(0, tperiod); /* temp 0, 100 ns */
1207	lm_message("Verify maximum skew between edges is 1000 ps max.\n");
1208	lm_message("Check all edges of all lanes of slot %d\n",
1209	tmg_play_til_key());
1210	
1211	
1212	
1213	/*-----long_tmg_continuouslocktest(void)
1214	/*
1215	This routine is a utility to allow measuring the PLL lock
1216	time on an oscilloscope or to display the PLL. The PLL
1217	is continuously slowed from 30 to 60 MHz and back.
1218	Inputs: none
1219	Outputs: after a key is pressed, always returns SUCCESS
1220	*/
1221	long_tmg_continuouslocktest()
1222	{
1223	if(tmg_clockoff() != SUCCESS) /* make sure clock is off */
1224	{
1225	(void)lm_error("tmg_continuouslocktest(): cannot turn clock off\n");
1226	return(FAILURE);
1227	}
1228	
1229	tmgptr->pll_rate = (256 - 128) + 1; /* sets to 30 MHz */
1230	tmgptr->pll_divisor = (256 - 15) + 1; /* choose div 2 + 15 */
1231	tmgptr->clock_select = 1; /* sel div 2K */
1232	
1233	lm_delay(2); /* wait lots of time for lock */
1234	
1235	(void)lm_message("You may observe TTLLOCK at U27-13 while triggering.\n");
1236	(void)lm_message("the oscilloscope with TRIGGER(7) at U160-10.\n");
1237	(void)lm_message("Press any key to abort...\n");
1238	while(!lm_check_key())
1239	{
1240	tmgptr->pll_rate = (256 - 256) + 1; /* set pll to 60 MHz */
1241	lm_delay(2); /* wait for lock */
1242	tmgptr->pll_rate = (256 - 128) + 1; /* set pll to 30 MHz */
1243	lm_delay(2);
1244	}
1245	return SUCCESS;
1246	}
1247	
1248	/*-----int tmg_clock_sync_test()
1249	/*
1250	This test checks that the synchronizer really does turn the clock
1251	on and off. It returns SUCCESS or FAILURE.
1252	*/
1253	int tmg_clock_sync_test()
1254	{
1255	long start;
1256	static char me[] = "tmg_clock_sync_test";
1257	
1258	tmgptr->clock_enable = 0; /* try to turn the clock off */
1259	start = lm_time();
1260	while(tmgptr->clock_on)
1261	{
1262	if((lm_time() - start) > 10)
1263	{
1264	if(tmgptr->clock_select > 2) /* must be external clock */
1265	tmgptr->clock_sync_clearl = 0; /* ok to clear sync */
1266	else
1267	{
1268	lm_error("%s: cannot clear synchronizer\n", me);
1269	return FAILURE;
1270	}
1271	}
1272	}
1273	tmgptr->clock_sync_clearl = 0; /* keep it cleared */
1274	tmgptr->clock_select = 0;
1275	tmgptr->pll_rate = 0x1c; /* set up for 15 MHz */
1276	tmgptr->pll_divisor = 0x1f; /* divide by 2 */
1277	
1278	tmgptr->clock_enable = 1; /* hope it doesn't come on */
1279	start = lm_time();
1280	while((lm_time() - start) > 10)
1281	{
1282	if(tmgptr->clock_on) /* clock came on, bad clock */
1283	return FAILURE;
1284	}
1285	tmgptr->clock_enable = 0;
1286	tmgptr->clock_sync_clearl = 1;
1287	start = lm_time();
1288	while((lm_time() - start) > 10)
1289	{
1290	if(tmgptr->clock_on) /* clock came on, bad clock */
1291	return FAILURE;
1292	}
1293	tmgptr->clock_enable = 1; /* this time it should come on */
1294	start = lm_time();
1295	while(!tmgptr->clock_on)
1296	{
1297	if((lm_time() - start) > 10)
1298	{
1299	lm_error("%s: timeout waiting for clock to turn on.\n", me);
1300	return FAILURE;
1301	}
1302	}
1303	tmgptr->clock_enable = 0; /* this time it should go off */
1304	start = lm_time();
1305	while(tmgptr->clock_on)
1306	{
1307	if((lm_time() - start) > 10)
1308	{
1309	lm_error("%s: timeout waiting for clock to turn on.\n", me);
1310	return FAILURE;
1311	}
1312	}
1313	tmgptr->clock_sync_clearl = 0; /* clear things and exit */
1314	return SUCCESS;
1315	}
1316	/*-----int tmg_dac_settle(void)
1317	/*
1318	This routine slows DAC 0 output back and forth from 0 to 255
1319	to allow measurement of slew rate and settling time on an
1320	*/

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89 PAGE #  
TME 4:41:33 pm 12/188

```

1321 * oscilloscope.
1322 *
1323 * Input: none
1324 * Output: always returns SUCCESS
1325 */
1326 tmg_dac_settle()
1327 {
1328     /* Define DAC (char *)0x00000001
1329     (void)lm_message("You may now observe ALOCIDAO at U61-9.\n");
1330     (void)lm_message("Press any key to abort...\n");
1331     while(!lm_check_key())
1332     {
1333         *DAC = 0;
1334         lm_delay(2);
1335         *DAC = 255;
1336         lm_delay(2);
1337     }
1338 }
1339 return SUCCESS;
1340 }
1341
1342 #ifndef BIN_LIMIT
1343 #define BIN_LIMIT 1000
1344 #endif
1345
1346 tmg_edge_jitter_search(ramp,edge,thr,upper_bound,lower_bound,direction)
1347 long ramp, edge, thr, upper_bound, lower_bound, direction;
1348 {
1349     static char me[] = "tmg_edge_jitter_search";
1350     long period, found_high;
1351     long timeout = BIN_LIMIT;
1352     long chk_period, save_lower_bound, save_upper_bound;
1353     long jitterptr, sptr, kptr, selectptr;
1354
1355     save_lower_bound = lower_bound;
1356     save_upper_bound = upper_bound;
1357     for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {
1358         period = (upper_bound + lower_bound)/2;
1359         if (tmg_detect_edge(ramp,edge,
1360             (direction ? DETECT_LOW : 0) | DETECT_BOOL, &period,
1361             thr,100001, &found_high) != SUCCESS) {
1362             tmg_report_failure(me, "tmg_detect_always_low(3)");
1363         }
1364         #ifdef DIAGS
1365             if (tmg_cal_delay & 1)
1366                 tmg_play_til_key();
1367         #endif DIAGS
1368         if ((period < save_lower_bound) || (period > save_upper_bound))
1369             lm_error("%s binary search failure(1)\n", me);
1370         return period;
1371     }
1372     if ((period <= lower_bound) || (period >= upper_bound)) {
1373         /* XXX */
1374         while (1) {
1375             tmg_get_next_lower_period(period, &chk_period,
1376                 &jitterptr,&sptr,&kptr,&selectptr);
1377             if (chk_period <= lower_bound) break;
1378             if (tmg_detect_edge(ramp,edge,
1379                 (direction ? DETECT_LOW : 0) |
1380                 DETECT_BOOL, &chk_period,
1381                 thr,100001, &found_high) != SUCCESS) {
1382                 tmg_report_failure(me,
1383                     "tmg_detect_edge(4)");
1384             }
1385             if (direction ? (found_high : found_high) {
1386                 /* too short - raise lower bound */
1387                 lower_bound = chk_period;
1388                 break;
1389             } else {
1390                 /* too long - lower upper bound */
1391                 upper_bound = chk_period;
1392             }
1393             period = chk_period;
1394         }
1395         /* We can't get more exact than this */
1396         #ifdef DIAGS
1397             if (tmg_cal_delay & 4)
1398                 lm_message("(got it between %d and %d)\n",
1399                     lower_bound, upper_bound);
1400         #endif DIAGS
1401         return (direction ? upper_bound : chk_period);
1402     }
1403     if (direction ? (found_high : found_high) {
1404         /* too short - raise lower bound */
1405         lower_bound = period;
1406     } else {
1407         /* too long - lower upper bound */
1408         upper_bound = period;
1409     }
1410     lm_error("%s binary search failure(2)\n", me);
1411     return period;
1412 } /* tmg_edge_jitter_search */
1413
1414 tmg_sample_jitter_search(eramp,eramp,thr7,thr,upper_bound,lower_bound,direction)
1415 long eramp, eramp, thr7, thr, upper_bound, lower_bound, direction;
1416 {
1417     static char me[] = "tmg_sample_jitter_search";
1418     long period, found_high;
1419     long timeout = BIN_LIMIT;
1420     long chk_period, save_lower_bound, save_upper_bound;
1421     long jitterptr, sptr, kptr, selectptr;
1422
1423     save_lower_bound = lower_bound;
1424     save_upper_bound = upper_bound;
1425     for (timeout = 0; timeout <= BIN_LIMIT; ++timeout) {
1426         period = (upper_bound + lower_bound)/2;
1427         if (tmg_detect_sample(eramp,eramp,
1428             (direction ? DETECT_LOW : 0) | DETECT_SAMPLE,
1429             (long*)&period, thr7,thr,100001, (long*)&found_high) != SUCCESS)
1430         {
1431             tmg_report_failure(me,

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89

PAGE #

TIME 4:41:33 pm

13/189

```

1441      "tmg_detect_sample(3)");
1442  }
1443  #ifdef DIAGS
1444      if (tmg_cal_debug & 0x100)
1445          tmg_play_til_key();
1446  #endif DIAGS
1447      if ((period < save_lower_bound) || (period > save_upper_bound))
1448      {
1449          lm_error("ta binary search failure(1)\n", me);
1450          return period;
1451      }
1452      if ((period <= lower_bound) || (period >= upper_bound)) {
1453          /* XXX */
1454          while (1) {
1455              tmg_get_next_lower_period(period, &chk_period,
1456                  &jitter_ptr, &ptr, &select_ptr);
1457              if (chk_period <= lower_bound) break;
1458              if (tmg_detect_sample(aramp, aramp,
1459                  (direction ? DETECT_LOW : 0) |
1460                  DETECT_BOOL | DETECT_SAMPLE,
1461                  (long*)&chk_period, &thr7, &thr8,
1462                  100001, (long*)&found_high)
1463                  != SUCCESS)
1464              {
1465                  tmg_report_failure(me,
1466                      "tmg_detect_sample(4)");
1467                  if (direction ? found_high : found_high) {
1468                      /* too short - raise lower bound */
1469                      lower_bound = chk_period;
1470                      break;
1471                  } else {
1472                      /* too long - lower upper bound */
1473                      upper_bound = chk_period;
1474                      period = chk_period;
1475                  }
1476              }
1477          }
1478          /* We can't get more exact than this */
1479      }
1480  #ifdef DIAGS
1481      if (tmg_cal_debug & 0x400)
1482          lm_message("(got it between %d and %d)\n",
1483              lower_bound, upper_bound);
1484  #endif DIAGS
1485      return (direction ? upper_bound : lower_bound);
1486  }
1487      if (direction ? found_high : found_high) {
1488          /* too short - raise lower bound */
1489          lower_bound = period;
1490      } else {
1491          /* too long - lower upper bound */
1492          upper_bound = period;
1493      }
1494      lm_error("ta binary search failure(2)\n", me);
1495      return period;
1496  }
1497  /* tmg_sample_jitter_search */
1498  }
1499  }
1500  }
1501  tmg_jitter_test()
1502  {
1503      int aramp, aramp, edge;
1504      long low_bound, high_bound, target, low_target, high_target;
1505      if (!calib.CalCompleted)
1506      {
1507          tmg_calibrate();
1508          if (!calib.CalCompleted)
1509          {
1510              lm_error("Cannot perform jitter test due to calibration failure\n");
1511              return FAILURE;
1512          }
1513      }
1514      tmg_set_test_mode(1);
1515      tmg_set_alot_count(2);
1516      tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE; /* select Edge7 trigger */
1517      for (aramp = 0; aramp < NUMBER_OF_EDGES; aramp++)
1518      {
1519          /* check jitter on all edges */
1520          for (edge = 0; edge < NUMBER_OF_EDGES; edge++)
1521          {
1522              target = tmg_predict_period(250, calib.Edgeslope[aramp][edge],
1523                  calib.EdgeOffset[aramp][edge]);
1524              low_target = 0.9 * target;
1525              high_target = 1.1 * target;
1526              low_bound = tmg_edge_jitter_search(aramp, edge, 250, high_target, low_target, 0);
1527              high_bound = tmg_edge_jitter_search(aramp, edge, 250, high_target, low_target, 1);
1528              lm_message("Ramp %d, Edge %d, target %d, found %d-%d, jitter %d\n",
1529                  aramp, edge, target, low_bound, high_bound, high_bound - low_bound);
1530          }
1531          /* do edge 7 */
1532          target = 250 * (long)calib.Edgeslope[aramp]
1533              + 10 * (long)calib.SampleOffset[0];
1534          low_target = 0.9 * target;
1535          high_target = 1.1 * target;
1536          low_bound = tmg_sample_jitter_search(aramp, 0, 250, 10,
1537              high_target, low_target, 0);
1538          high_bound = tmg_sample_jitter_search(aramp, 0, 250, 10,
1539              high_target, low_target, 1);
1540          lm_message("Ramp %d, sample, target %d, found %d-%d, jitter %d\n",
1541              aramp, target, low_bound, high_bound, high_bound - low_bound);
1542      }
1543      /* now do sample */
1544      tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE; /* select early sample */
1545      for (aramp = 0; aramp < NUMBER_OF_EDGES; aramp++)
1546      {
1547          target = 255 * (long)calib.SampleSlope[aramp]
1548              + (long)calib.EarlySampleOffset[aramp];
1549          low_target = 0.9 * target;
1550          high_target = 1.1 * target;
1551          low_bound = tmg_sample_jitter_search(1, aramp, 255, 255,
1552              high_target, low_target, 0);
1553          high_bound = tmg_sample_jitter_search(1, aramp, 255, 255,
1554              high_target, low_target, 1);
1555          lm_message("Ramp %d, sample, target %d, found %d-%d, jitter %d\n",
1556              aramp, target, low_bound, high_bound, high_bound - low_bound);
1557      }
1558  }
1559  }
1560  }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_diag.c

DATE 5/23/89 PAGE #  
TIME 4:41:33 pm 14/190

```

1561 return SUCCESS,
1562 }
1563
1564
1565
1566
1567 /* This function sets the period to 128 ns. It takes each edge in turn
1568 * and places it at 50 ns. It then takes all other (2) edges that share
1569 * the same delay line module and sweeps them from 30 ns to 70 ns. If
1570 * any coupling is detected, the function fails.
1571 */
1572 tmg_delay_line_isolation()
1573 {
1574     int target_edge, errors, count;
1575     long period;
1576
1577     errors = 0;
1578     tmgptr->backplane_reset = 0; /* assert backplane reset */
1579     tmg_set_slot_count(1);
1580     tmg_set_test_mode(1);
1581     tmg_set_aramp(0);
1582     period = 128000L;
1583     if(tmg_set_period(0, &period) != SUCCESS)
1584     {
1585         lm_error("tmg_delay_line_isolation: tmg_set_period()");
1586         return FAILURE;
1587     }
1588
1589     for(target_edge = 0; target_edge < 6; target_edge++)
1590     {
1591         tmgptr->edge_delay[target_edge].delay = tmg_predict_threshold(50000L,
1592             calib.EdgeSlope[0][target_edge],
1593             calib.EdgeOffset[0][target_edge]);
1594
1595         for(count = 1000; count > 0; count--)
1596             switch (target_edge)
1597             {
1598                 case 0:
1599                     tmg_sweep(target_edge, 1, &errors, &calib);
1600                     tmg_sweep(target_edge, 2, &errors, &calib);
1601                     break;
1602                 case 1:
1603                     tmg_sweep(target_edge, 0, &errors, &calib);
1604                     tmg_sweep(target_edge, 2, &errors, &calib);
1605                     break;
1606                 case 2:
1607                     tmg_sweep(target_edge, 0, &errors, &calib);
1608                     tmg_sweep(target_edge, 1, &errors, &calib);
1609                     break;
1610                 case 3:
1611                     tmg_sweep(target_edge, 4, &errors, &calib);
1612                     tmg_sweep(target_edge, 5, &errors, &calib);
1613                     break;
1614                 case 4:
1615                     tmg_sweep(target_edge, 3, &errors, &calib);
1616                     tmg_sweep(target_edge, 5, &errors, &calib);
1617                     break;
1618                 case 5:
1619                     tmg_sweep(target_edge, 3, &errors, &calib);
1620                     tmg_sweep(target_edge, 4, &errors, &calib);
1621                     break;
1622                 default:
1623                     break;
1624             }
1625
1626         (void)diag_tmg_reset(1); /* Full reset */
1627         if(errors)
1628             return FAILURE;
1629         else
1630             return SUCCESS;
1631     }
1632
1633     tmg_sweep(target_edge, other_edge, &errorptr, &calptr)
1634     int target_edge, other_edge, *errorptr;
1635     struct CALIB *calptr;
1636
1637     int thresh, current_errors, actual, count, min, max;
1638     current_errors = 0;
1639     min = tmg_predict_threshold(50000L,
1640         calptr->EdgeSlope[0][other_edge],
1641         calptr->EdgeOffset[0][other_edge]);
1642     max = tmg_predict_threshold(70000L,
1643         calptr->EdgeSlope[0][other_edge],
1644         calptr->EdgeOffset[0][other_edge]);
1645     for(thresh = min; thresh < max; thresh++)
1646     {
1647         tmgptr->edge_delay[other_edge].delay = thresh;
1648         for(count = 1000; count > 0; count--)
1649             if (tmg_detect(target_edge, 100L, &actual, DETECT_BOOL) != SUCCESS)
1650             {
1651                 current_errors++;
1652                 break;
1653             }
1654             if(actual == 0)
1655             {
1656                 current_errors++;
1657                 break;
1658             }
1659     }
1660
1661     if(current_errors)
1662     {
1663         *errorptr += current_errors;
1664         lm_message("delay line coupling detected between edges %d and %d\n",
1665             target_edge, other_edge);
1666     }
1667 }
1668

```

Copyright 1989 Logic Modeling Systems		HEADER FILE diags/tmg_extn.h	DATE 5/23/89	PAGE # 1/191
			TIME 4:41:35 pm	
LINE #	HEADER TEXT			
1	/* SOCS_ID: tmg_extn.h rev 3.1.1, 4/24/89 at 07:50:42 */			
2	/*			
3	.....			
4	tmg_extn.h			
5	.....			
6	External declarations of global variables for the Timing			
7	Generator Diagnostics and Calibration			
8	Last Modified 7/18/88 -- wfk			
9	.....			
10	*/			
11	.....			
12	extern TMC *tmcptr;			
13	extern TMC_DIAG *tmcdiagptr;			
14	extern char mbr[[]];			
15	extern int intflag;			
16	extern struct CALLS calib;			
17	extern u_char *tmg_sptr;			
18	extern u_char *tmg_lptr;			
19	extern u_char *tmg_arcptr;			
20	extern u_long *tmg_periodptr;			
21	extern u_long tmg_max;			
22	extern int tmg_freq_table_built;			
23	extern u_long tmg_min_index1[];			
24	extern u_long tmg_max_index1[];			
25	extern u_long tmg_min_index2[];			
26	extern u_long tmg_max_index2[];			
27	extern u_long tmg_min_index3[];			
28	extern u_long tmg_max_index3[];			
29	extern u_long tmg_min_index4[];			
30	extern u_long tmg_max_index4[];			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_glbl.c	DATE 5/23/89	PAGE # 1/192
LINE #	SOURCE TEXT			
1	/* GCCS_ID: tmg_glbl.c rev 1.1, 4/24/89 at 07:50:50 */			
2	/*.....			
3	/*			
4	/* tmg_glbl.c			
5	/*			
6	/* Global variables used in Timing Generator			
7	/* Diagnostics and Calibration.			
8	/*			
9	/*			
10	/*.....			
11	/*			
12	#include "common.h"			
13	#include "tmg.h"			
14	#include "tmg_def.h"			
15	/*			
16	TMC *tmgptr = (TMC *)CLOCK_BASE; /* pointer to TMC */			
17	TMC_DIAG *tmgdiagptr = (TMC_DIAG *)CLOCK_BASE; /* alternate pointer */			
18	/*			
19	/* global variables used for generating/maintaining frequency table */			
20	u_char *tmg_ptr; /* pointer to k register values */			
21	u_char *tmg_kptr; /* pointer to k register values */			
22	u_char *tmg_sptr; /* pointer to clock select values */			
23	u_long *tmg_periodptr; /* pointer to clock period values */			
24	u_long *tmg_max; /* pointer to maximum index for above */			
25	int tmg_freq_table_built = FALSE;			
26	u_long tmg_min_index[4]; /* range limits for searching during */			
27	u_long tmg_max_index[4]; /* calibration */			
28	u_long tmg_min_index2[4];			
29	u_long tmg_max_index2[4];			
30	u_long tmg_min_index3[4];			
31	u_long tmg_max_index3[4];			
32	/*			
33	struct CALIB calib =			
34	{			
35	{			
36	{			
37	{			
38	{			
39	{			
40	{			
41	{10000, 16000, 80000, 400000}, /* EdgeMinThresh */			
42	{128000, 512000, 2560000, 12800000}, /* EdgeMinDelay */			
43	{			
44	{0, 0, 0, 0, 0, 0},			
45	{0, 0, 0, 0, 0, 0},			
46	{0, 0, 0, 0, 0, 0},			
47	{0, 0, 0, 0, 0, 0},			
48	{			
49	{575, 575, 575, 575, 575, 575},			
50	{2300, 2300, 2300, 2300, 2300, 2300},			
51	{11500, 11500, 11500, 11500, 11500, 11500},			
52	{57500, 57500, 57500, 57500, 57500, 57500},			
53	{			
54	{			
55	{0, 0, 0, 0, 0, 0},			
56	{0, 0, 0, 0, 0, 0},			
57	{0, 0, 0, 0, 0, 0},			
58	{0, 0, 0, 0, 0, 0},			
59	{			
60	{			
61	{575, 2300, 11500, 57500},			
62	{0, 0, 0, 0},			
63	{8, 8, 8, 8},			
64	{150000, 62000, 126000, 446000},			
65	{128000, 256000, 512000, 1024000},			
66	{500, 1000, 2000, 4000},			
67	{0, 0, 0, 0},			
68	{			
69	{30000, 30000, 30000, 30000},			
70	{30000, 30000, 30000, 30000},			
71	{30000, 30000, 30000, 30000},			
72	{30000, 30000, 30000, 30000},			
73	{30000, 30000, 30000, 30000},			
74	{			
75	#ifdef EARLYSAMPLETRIGGER			
76	{9000, 13000, 21000, 17000},			
77	{110000, 220000, 440000, 880000},			
78	{0, 0, 0, 0},			
79	#endif			
80	{0, 0, 0, 0},			
81	{			
82	{			
83	{			
84	{			
85	struct CALIB default_calib =			
86	{			
87	{			
88	{			
89	{			
90	{			
91	{			
92	{			
93	{10000, 16000, 80000, 400000}, /* EdgeMinThresh */			
94	{128000, 512000, 2560000, 12800000}, /* EdgeMinDelay */			
95	{			
96	{0, 0, 0, 0, 0, 0},			
97	{0, 0, 0, 0, 0, 0},			
98	{0, 0, 0, 0, 0, 0},			
99	{0, 0, 0, 0, 0, 0},			
100	{			
101	{575, 575, 575, 575, 575, 575},			
102	{2300, 2300, 2300, 2300, 2300, 2300},			
103	{11500, 11500, 11500, 11500, 11500, 11500},			
104	{57500, 57500, 57500, 57500, 57500, 57500},			
105	{			
106	{			
107	{0, 0, 0, 0, 0, 0},			
108	{0, 0, 0, 0, 0, 0},			
109	{0, 0, 0, 0, 0, 0},			
110	{0, 0, 0, 0, 0, 0},			
111	{			
112	{			
113	{575, 2300, 11500, 57500},			
114	{0, 0, 0, 0},			
115	{8, 8, 8, 8},			
116	{150000, 62000, 126000, 446000},			
117	{128000, 256000, 512000, 1024000},			
118	{500, 1000, 2000, 4000},			
119	{0, 0, 0, 0},			
120	{			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/trng_glbl.c	DATE 5/23/89	PAGE # 2/193
LINE #	SOURCE TEXT			
121	{			
122	{30000, 30000, 30000, 30000},			
123	{30000, 30000, 30000, 30000},			
124	{30000, 30000, 30000, 30000},			
125	{30000, 30000, 30000, 30000}			
126	/* SampleOffset */			
127	/* EarlySampleTrigger			
128	{9000, 13000, 21000, 37000},			
129	{110000, 220000, 440000, 880000},			
130	{0, 0, 0, 0},			
131	/* EarlySampleOffset */			
132	/* DelayDelay			
133	/* DumpDelay			
134	/* CalCompleted			
135	};			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_menu.c

DATE 5/23/89 PAGE #  
TIME 4:41:35 pm 1/194

LINE # SOURCE TEXT

1 /\* SCCS ID: tmg\_menu.c rev 3.2, 5/9/89 at 15:55:01 \*/

2 /\*.....\*/

3 /\*.....\*/

4 /\*.....\*/

5 /\*.....\*/

6 /\*.....\*/

7 /\*.....\*/

8 /\*.....\*/

9 /\*.....\*/

10 /\*.....\*/

11 #include <math.h>

12 #include "common.h"

13 #include "tmg.h"

14 #include "tmg\_def.h"

15 #include "vrtx.h"

16 #include "lm\_diags.h"

17 #include "tmg\_exts.h"

18

19 tmg\_diag\_disp(parent\_menu)

20 LM\_DIAG\_MENU "parent\_menu,

21 {

22 extern int tmg\_register\_test(),

23 tmg\_idprom\_test(),

24 tmg\_clock\_sync\_test(),

25 tmg\_etc\_test(),

26 tmg\_pll\_rate(),

27 tmg\_pll\_div(),

28 tmg\_pll\_lock(),

29 tmg\_slow\_detect(),

30 tmg\_interrupts(),

31 tmg\_calibrate(),

32 tmg\_diag\_disp2(),

33 tmg\_delay\_line\_isolation();

34

35 static LM\_DIAG\_MENU\_ITEM menu\_list[] =

36 {

37 {

38 "1",

39 "Register Test",

40 tmg\_register\_test,

41 LM\_DIAG\_diag\_routine,

42 LM\_DIAG\_null

43 },

44 {

45 "2",

46 "ID Prom Test",

47 tmg\_idprom\_test,

48 LM\_DIAG\_diag\_routine,

49 LM\_DIAG\_null

50 },

51 {

52 "3",

53 "Clock Synchronizer Test",

54 tmg\_clock\_sync\_test,

55 LM\_DIAG\_diag\_routine,

56 LM\_DIAG\_null

57 },

58 {

59 "4",

60 "Counter Timer Test",

61 tmg\_etc\_test,

62 LM\_DIAG\_diag\_routine,

63 LM\_DIAG\_null

64 },

65 {

66 "5",

67 "Phase Locked Loop Rate",

68 tmg\_pll\_rate,

69 LM\_DIAG\_diag\_routine,

70 LM\_DIAG\_null

71 },

72 {

73 "6",

74 "Phase Locked Loop Division",

75 tmg\_pll\_div,

76 LM\_DIAG\_diag\_routine,

77 LM\_DIAG\_null

78 },

79 {

80 "7",

81 "Phase Locked Loop Lock Time",

82 tmg\_pll\_lock,

83 LM\_DIAG\_diag\_routine,

84 LM\_DIAG\_null

85 },

86 {

87 "8",

88 "Slow Clock Detector Test",

89 tmg\_slow\_detect,

90 LM\_DIAG\_diag\_routine,

91 LM\_DIAG\_null

92 },

93 {

94 "9",

95 "Check Interrupt Sources",

96 tmg\_interrupts,

97 LM\_DIAG\_diag\_routine,

98 LM\_DIAG\_null

99 },

100 {

101 "10",

102 "Check Calibration",

103 tmg\_calibrate,

104 LM\_DIAG\_diag\_routine,

105 LM\_DIAG\_null

106 },

107 {

108 "11",

109 "Check Edge Isolation",

110 tmg\_delay\_line\_isolation,

111 LM\_DIAG\_diag\_routine,

112 LM\_DIAG\_null

113 },

114 {

115 "12",

116 "Utilities",

117 tmg\_diag\_disp2,

118 LM\_DIAG\_utility\_menu,

119 LM\_DIAG\_null

120 }

Copyright 1989  
Logic Modeling SystemsSOURCE PROGRAM  
diags/tmg\_menu.c

\*

DATE 5/23/89  
TIME 4:41:35 pmPAGE #  
2/195

LINE # SOURCE TEXT

```
121  },
122  static LM_DIAG_MENU menu =
123  {
124    "TIMING GENERATOR DIAGNOSTICS",
125    sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
126    0,
127    menu_list
128  },
129  },
130  menu.title = parent_menu->
131  menu.items[parent_menu->current_selection].menu_text,
132  },
133  return lm_display_menu(&menu);
134  }
135  }
136  tmg_diag_disp2(parent_menu)
137  LM_DIAG_MENU *parent_menu;
138  {
139    int tmg_freq_ext0();
140    int tmg_freq_ext1();
141    int pac_sel_pat_clk();
142    int tmg_select_mode();
143    int tmg_select_ramps();
144    int tmg_select_thresholds();
145    int tmg_select_slot_count();
146    int tmg_loop();
147    int print_calib_structure();
148    int tmg_plot_delays();
149    int tmg_modify_debug_flag();
150    int tmg_jitter_test();
151    int tmg_scope_measurements();
152  }
153  static LM_DIAG_MENU_ITEM menu_list[] =
154  {
155    {
156      "1",
157      "Measure Ext0 Frequency",
158      tmg_freq_ext0,
159      LM_DIAG_utility,
160      LM_DIAG_null
161    },
156    {
162      "2",
163      "Measure Ext1 Frequency",
164      tmg_freq_ext1,
165      LM_DIAG_utility,
166      LM_DIAG_null
167    },
168    {
169      "3",
170      "Select Pattern Clock Period",
171      pac_sel_pat_clk,
172      LM_DIAG_utility,
173      LM_DIAG_null
174    },
175    {
176      "4",
177      "Select Test/Sample Modes",
178      tmg_select_mode,
179      LM_DIAG_utility,
180      LM_DIAG_null
181    },
182    {
183      "5",
184      "Select Edge/Sample Ramps",
185      tmg_select_ramps,
186      LM_DIAG_utility,
187      LM_DIAG_null
188    },
189    {
190      "6",
191      "Select Edge/Sample Thresholds",
192      tmg_select_thresholds,
193      LM_DIAG_utility,
194      LM_DIAG_null
195    },
196    {
197      "7",
198      "Select Slot Count",
199      tmg_select_slot_count,
200      LM_DIAG_utility,
201      LM_DIAG_null
202    },
203    {
204      "8",
205      "PLAY",
206      tmg_loop,
207      LM_DIAG_utility & "LM_DIAG_no_repeat",
208      LM_DIAG_null
209    },
210    {
211      "9",
212      "Print Current Calibration Structure",
213      print_calib_structure,
214      LM_DIAG_utility,
215      LM_DIAG_null
216    },
217    {
218      "10",
219      "Plot Delay Line Delay vs Clock Frequency",
220      tmg_plot_delays,
221      LM_DIAG_utility,
222      LM_DIAG_null
223    },
224    {
225      "11",
226      "Enable/Disable Optional Debugging Messages Menu",
227      tmg_modify_debug_flag,
228      LM_DIAG_utility_menu,
229      LM_DIAG_null
230    },
231    {
232      "12",
233      "Jitter Measurement",
234      tmg_jitter_test,
235      LM_DIAG_utility,
236      LM_DIAG_null
237    },
238    {
239      "13",
240  }
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_menu.c	DATE 5/23/89 TIME 4:41:35 pm	PAGE # 3/196
LINE #	SOURCE TEXT			
241	"Oscilloscope Measurements Menu",			
242	tmg_scope_measurements,			
243	LM_DIAG_utility_menu,			
244	LM_DIAG_null			
245	},			
246	static LM_DIAG_MENU menu =			
247	{			
248	"TIMING GENERATOR UTILITIES",			
249	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),			
250	0,			
251	menu_list			
252	},			
253	menu.title = parent_menu->			
254	menu_items[parent_menu->current_selection].menu_text,			
255	return lm_display_menu(&menu);			
256	}			
257	tmg_modify_debug_flag(parent_menu)			
258	LM_DIAG_MENU *parent_menu,			
259	{			
260	extern int tmg_modify_edge_related_debug_flag(),			
261	tmg_modify_sample_related_debug_flag(),			
262	tmg_modify_delayline_related_debug_flag(),			
263	tmg_modify_misc_related_debug_flag(),			
264	tmg_modify_all_debug_flag(),			
265	static LM_DIAG_MENU_ITEM menu_list[] =			
266	{			
267	"1",			
268	"Edge Calibration Related Debug Flags",			
269	tmg_modify_edge_related_debug_flag,			
270	LM_DIAG_utility   LM_DIAG_automatic_quit,			
271	LM_DIAG_null			
272	},			
273	"2",			
274	"Sample Calibration Related Debug Flags",			
275	tmg_modify_sample_related_debug_flag,			
276	LM_DIAG_utility   LM_DIAG_automatic_quit,			
277	LM_DIAG_null			
278	},			
279	"3",			
280	"Delay Line Calibration Related Debug Flags",			
281	tmg_modify_delayline_related_debug_flag,			
282	LM_DIAG_utility   LM_DIAG_automatic_quit,			
283	LM_DIAG_null			
284	},			
285	"4",			
286	"Miscellaneous Calibration Related Debug Flags",			
287	tmg_modify_misc_related_debug_flag,			
288	LM_DIAG_utility   LM_DIAG_automatic_quit,			
289	LM_DIAG_null			
290	},			
291	"5",			
292	"Set/Reset All Debug Flags",			
293	tmg_modify_all_debug_flag,			
294	LM_DIAG_utility   LM_DIAG_automatic_quit,			
295	LM_DIAG_null			
296	},			
297	static LM_DIAG_MENU menu =			
298	{			
299	"TIMING GENERATOR DEBUG FLAGS",			
300	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),			
301	0,			
302	menu_list			
303	},			
304	menu.title = parent_menu->			
305	menu_items[parent_menu->current_selection].menu_text,			
306	return(lm_display_menu(&menu));			
307	}			
308	tmg_scope_measurements(parent_menu)			
309	LM_DIAG_MENU *parent_menu,			
310	{			
311	int tmg_continuouslocktest(),			
312	int tmg_dac_settle(),			
313	int tmg_edge_jitter(),			
314	int tmg_sample_jitter(),			
315	int tmg_edge_alignment(),			
316	static LM_DIAG_MENU_ITEM menu_list[] =			
317	{			
318	"1",			
319	"Continuous Phase Lock Loop Lock/Unlock",			
320	tmg_continuouslocktest,			
321	LM_DIAG_utility,			
322	LM_DIAG_null			
323	},			
324	"2",			
325	"D/A Converter Settling Time",			
326	tmg_dac_settle,			
327	LM_DIAG_utility,			
328	LM_DIAG_null			
329	},			
330	"3",			
331	"Edge Jitter Measurement",			
332	tmg_edge_jitter,			
333	LM_DIAG_utility,			
334	LM_DIAG_null			
335	},			
336	"4",			
337	"Sample Jitter Measurement",			
338	tmg_sample_jitter,			
339	LM_DIAG_utility,			
340	LM_DIAG_null			
341	},			
342	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_menu.c

DATE 5/23/89  
TIME 4:41:35 pm

PAGE #  
4/197

LINE #	SOURCE TEXT
361	{
362	"5",
363	"Edge Time Alignment",
364	tmg_edge_alignment,
365	LM_DIAG_utility,
366	LM_DIAG_null
367	}
368	},
369	
370	static LM_DIAG_MENU menu =
371	{
372	"TIMING GENERATOR OSCILLOSCOPE MEASUREMENTS",
373	sizeof(menu_list) / sizeof(LM_DIAG_MENU_ITEM),
374	0,
375	menu_list
376	}
377	menu.title = parent_menu->
378	menu_items[parent_menu->current_selection].menu_text,
379	
380	return lm_display_menu(&menu);
381	}
382	



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

1/198

```

1  /* SOCS_ID: tmg_run.c rev 1.1, 4/24/89 at 07:50:56 */
2  /*-----*/
3  /*
4  *   tmg_run.c
5  *   Timing Generator Run-time support routines
6  *   -----*/
7  /*-----*/
8  /*
9  *   Includes
10  *   -----*/
11  #include "common.h"
12  #include "tmg.h"
13  #include "message.h"
14  #include "tmg_def.h"
15  #include "tmg_run.h"
16  #include "vtr.h"
17  #include "math.h"
18  /*-----*/
19  extern u_char play_completed_flag;
20  u_char post_end_of_play;
21  extern int play_semaphore;
22  static int zero = 0;
23  /*-----*/
24  #define DEBUG
25  /*-----*/
26  #ifdef DEBUG
27  #define DPRINTF(x) printf x
28  #else
29  #define DPRINTF(x) /* do nothing */
30  #endif
31  /*-----*/
32  /*
33  *   In tmg_get_frequency_setting()
34  *   -----*/
35  /*
36  *   This routine computes the values for n, k, and clock
37  *   select register setting to give the best approximation
38  *   of the desired clock period. The clock period provided
39  *   is always equal to or greater than that requested.
40  *   The actual clock period is related to n, k as follows:
41  *   period = (k / n) * T_REF, where
42  *   k = 1, 2, ..., 256 or k = 2, 4, ..., 512,
43  *   n = 128, 129, ..., 256, and
44  *   T_REF = 256 / 30 MHz = PLL reference period.
45  *   Returns: FAILURE if period out of range, else SUCCESS
46  *   -----*/
47  /*
48  *   In tmg_get_frequency_setting(period, actualptr, jitterptr, sptr, kptr, selectptr)
49  *   -----*/
50  u_long period; /* physical clock period in ps */
51  u_long *actualptr; /* actual clock period in ps */
52  u_long *jitterptr; /* jitter per period in ps */
53  u_long *sptr; /* select register */
54  u_long *kptr; /* k */
55  u_long *selectptr; /* select register */
56  {
57  double error, besterror, kprime;
58  int tmpk, n, save_k, save_n;
59  /*-----*/
60  if((period > MAX_PERIOD) || (period < MIN_PERIOD)) {
61  /* queue message(ERROR_MSG, "period out of range, period = %d",
62  period);
63  return(FAILURE);
64  }
65  /*-----*/
66  for(besterror = 1.0, n = N_MIN, k = N_MAX, n++)
67  {
68  kprime = n * period / T_REF;
69  if((tmpk = (int)ceil(kprime)) > 512) /* see if out of range */
70  continue; /* if so, ignore it */
71  if(tmpk > 256) /* see if above 256 */
72  if(tmpk & 1) /* see if odd */
73  tmpk++; /* make even */
74  if(error = (((double)tmpk - kprime) / kprime) < besterror)
75  {
76  save_n = n;
77  save_k = tmpk;
78  besterror = error;
79  }
80  }
81  *sptr = 256 - save_n + 1; /* value to put in reg */
82  if(save_k == 1)
83  {
84  *kptr = 255; /* special case */
85  *selectptr = 0; /* select div by 2 */
86  }
87  else
88  if(save_k > 256)
89  {
90  *kptr = 256 - (save_k >> 1) + 1; /* divide k by 2 */
91  *selectptr = 2; /* select div by 4k */
92  }
93  else
94  {
95  *kptr = 256 - save_k + 1; /* normal case */
96  *selectptr = 1; /* select div by 2k */
97  }
98  *actualptr = T_REF * save_k / save_n;
99  *jitterptr = 25 * 2 * save_k; /* 20 ps per PLL clock */
100  return(SUCCESS);
101  }
102  /*-----*/
103  /*
104  *   In tmg_get_sample_ramp_delay()
105  *   -----*/
106  /*
107  *   This function returns delay of SAMPLE with the given threshold.
108  *   This contains an offset error and should not be used to compute
109  *   absolute delays. Using this function to compute a difference
110  *   in time is OK since the offset errors cancel. Delays are in ps.
111  *   Inputs: range (1 of 4 ramp rates), threshold value, pointer to
112  *   delay variable, pointer to error
113  *   Outputs: delay and error are assigned, function returns
114  *   SUCCESS or FAILURE
115  *   -----*/
116  /*
117  *   In tmg_get_sample_ramp_delay(rate, threshold, delayptr, errorptr)
118  *   -----*/
119  u_char rate;
120  long *delayptr, *errorptr;
121  }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_run.c	DATE 5/23/89	PAGE # 2/199
LINE #		SOURCE TEXT		
121		if((rate > 3)    (threshold < calib.samplesinthresh(rate))) {		
122		in_queue_message(ERROR_MSG, "rate too large OR threshold too small");		
123		return(FAILURE);		
124		}		
125		errorptr = 0;                   /* tentatively not used */		
126		if(threshold > sample_size)		
127		{		
128		*delayptr = threshold + calib.sampleslope(rate);		
129		/* errorptr = calib.sampleslope(rate); */		
130		}		
131		else		
132		{		
133		*delayptr = threshold + calib.sampleslope(rate)		
134		+ calib.earlysampleoffset(rate);		
135		/* errorptr = calib.earlysampleslope(rate) + 750; */		
136		}		
137		return(SUCCESS);		
138		}		
139		/*		
140		in_tmg_get_early_sample_thresh()		
141		/*		
142		This function returns threshold of early SAMPLE gives the delay.		
143		Inputs: range (1 of 4 ramp rates), delay value, pointer to		
144		threshold variable		
145		Outputs: threshold is assigned, function returns		
146		SUCCESS or FAILURE		
147		*/		
148		in_tmg_get_early_sample_thresh(rate, delay, thresholdptr)		
149		u_char *rate;		
150		u_long delay;		
151		u_char *thresholdptr;		
152		{		
153		u_long max_delay;		
154		max_delay = 255 + calib.sampleslope(rate) +		
155		calib.earlysampleoffset(rate);		
156		}		
157		if((rate > 3)    (delay > max_delay)) {		
158		in_queue_message(ERROR_MSG, "rate too large OR delay too large");		
159		return(FAILURE);		
160		}		
161		/*		
162		thresholdptr = (delay - calib.earlysampleoffset(rate)) /		
163		calib.sampleslope(rate);		
164		/*		
165		return(SUCCESS);		
166		/*		
167		} in_tmg_get_dead_time()		
168		/*		
169		This function returns dead time in ps for specified clock period		
170		in ps. Currently, this function returns dead time that depends		
171		only upon the EDGE 0 slope of ramp needed to open the clock period.		
172		Start dead time is due to non-zero minimum thresholds for the		
173		ladder ramp settings. Ending dead time is due to finite dump time		
174		of the ramp.		
175		/*		
176		Inputs: period in ps, pointers to start and ending dead times		
177		Outputs: dead times are assigned, function returns SUCCESS or FAILURE		
178		/*		
179		in_tmg_get_dead_time(period, startdeadptr, enddeadptr)		
180		u_long period;		
181		u_long *startdeadptr;		
182		u_long *enddeadptr;		
183		{		
184		register u_long ramp;		
185		}		
186		if((period < (u_long)MIN_PERIOD)    (period > (u_long)MAX_PERIOD)) {		
187		in_queue_message(ERROR_MSG, "period out of range; period = %d",		
188		period);		
189		return(FAILURE);		
190		}		
191		for(ramp = 0; ramp < 4; ramp++)		
192		{		
193		if(calib.EdgesMaxDelay[ramp] > period)		
194		{		
195		in_tmg_get_ramp_dead_time(period, ramp, startdeadptr, enddeadptr);		
196		return(SUCCESS);		
197		}		
198		}		
199		in_queue_message(ERROR_MSG, "period not reached; period = %d, EdgesMaxDelay[3] = %d",		
200		period, calib.EdgesMaxDelay[3]);		
201		return(FAILURE);		
202		}		
203		/*		
204		in_tmg_get_ramp_dead_time(period, ramp, startdeadptr, enddeadptr)		
205		register u_long period, ramp;		
206		u_long *startdeadptr;		
207		u_long *enddeadptr;		
208		{		
209		int i;		
210		long min_offset;		
211		}		
212		/* start dead time = edge minimum delay		
213		/*		
214		startdeadptr = calib.EdgesMinDelay[ramp];		
215		/*		
216		end dead time = ramp dump time		
217		/*		
218		+ ramp uncertainty		
219		/*		
220		+ 2 * edge jitter		
221		/*		
222		= min of all offsets of ramp 0		
223		/* Note: this should also include clock jitter, but		
224		/* this routine is unaware of clock frequency.		
225		/* This is accounted for elsewhere in runtime code		
226		/*		
227		min_offset = calib.EdgesOffset[0][0];		
228		for(i = 1; i < NUMBER_OF_EDGES; i++)		
229		{		
230		if(min_offset > calib.EdgesOffset[0][i])		
231		min_offset = calib.EdgesOffset[0][i];		
232		}		
233		enddeadptr = (long)RAMP_DUMP_TIME		
234		+ ((long)period / 501)		
235		+ 2 * calib.EdgesSlope[ramp][0]		
236		+ 2 * calib.EdgesSlope[ramp][0]		
237		- min_offset;		
238		/* uncertainty = 24		
239		/* + 2 thresholds		
240		/* + 2 * edge jitter		
241		/*		
242		return(SUCCESS);		
243		}		
244		/*		
245		in_tmg_get_falling_sample_setting()		

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

3/200

```

241 *
242 * This function returns the value to load into Timing Generator
243 * Register 7. The delay amount is equal to or greater than that
244 * requested. Software must guarantee that this delay is greater
245 * than the delay for the leading edge of SAMPLE.
246 *
247 * Inputs: period of clock in ps, delay requested in ps (referenced
248 * from last pattern), pointer to width setting
249 * Outputs: width is assigned, if delay is possible, function returns
250 * SUCCESS, else FAILURE
251 */
252
253 int tmg_get_falling_sample_setting(period, delay, widthptr)
254     u_long period;
255     u_long delay;
256     u_long *widthptr;
257 {
258     unsigned int tmp;
259
260     if((tmp = (unsigned int)((double)delay / (double)period)) > 255) {
261         /* requested delay of td ps requires td (>255) clocks,
262            delay, tmp;
263         */
264         return(FAILURE);
265     }
266     else
267     {
268         *widthptr = tmp;
269         return(SUCCESS);
270     }
271 }
272
273 /*
274 * int tmg_get_edge_setting()
275 *
276 * This function computes edge settings and selects appropriate
277 * ramp (edge 0) slope. All delays are less than or equal to
278 * what is requested.
279 *
280 * Inputs: logical clock period in ps, Array of desired edge
281 *         delay times in ps, pointer to array of threshold
282 *         settings to generate desired delays
283 * Outputs: threshold array is assigned along with quantization
284 *         and extra delay, function returns SUCCESS or FAILURE
285 */
286
287 int tmg_get_edge_setting(period, delay, threshold)
288     u_long period;
289     u_long delay[]; /* better be size 5 */
290     u_long threshold[];
291 {
292     int i, j;
293     u_long start, end, thresh;
294
295     if(!tmg_get_dead_time(period, &start, &end) != SUCCESS) {
296         return(FAILURE);
297     }
298
299     for(i = 0; i < 4; i++)
300     {
301         if(calib.EdgeMaxDelay[i] > period)
302             break;
303     }
304     if(i == 4) {
305         /* period too large, period = td, EdgeMaxDelay[3] = td,
306            period, calib.EdgeMaxDelay[3];
307         */
308         return(FAILURE);
309     }
310
311     for(j = 0; j < 6; j++)
312     {
313         if((delay[j] < calib.EdgeMinDelay[i]) || (delay[j] > (period - end))) {
314             /* delay out of range, delay[td] = td, EdgeMinDelay = td, max = td, n,
315                delay[j], calib.EdgeMinDelay[i], (period - end);
316             */
317             return(FAILURE);
318         }
319         threshold[j] = (delay[j] - calib.EdgeOffset[i][j]) / calib.Edgeslope[i][0];
320     }
321
322     return(SUCCESS);
323 }
324
325 /*
326 * int tmg_get_sample_setting()
327 *
328 * This function computes edge 7 settings.
329 *
330 * Inputs: logical clock period in ps, desired sample delay
331 *         time in ps, pointer to threshold setting to generate
332 *         desired delay.
333 * Outputs: threshold is assigned, function returns SUCCESS or FAILURE
334 */
335
336 int tmg_get_sample_setting(period, delay, thresholdptr)
337     u_long period;
338     u_long delay;
339     u_long *thresholdptr;
340 {
341     int eramp, edge, aramp;
342     u_long start, end;
343     long thresh;
344     long max_thresh;
345
346     if(!tmg_get_dead_time(period, &start, &end) != SUCCESS)
347         return(FAILURE);
348
349     for(eramp = 0; eramp < 4; eramp++)
350     {
351         if(calib.EdgeMaxDelay[eramp] > period)
352             break;
353     }
354     if(eramp == 4) {
355         /* period too large, period = td, EdgeMaxDelay[3] = td,
356            period, calib.EdgeMaxDelay[3];
357         */
358         return(FAILURE);
359     }
360
361     /* find maximum of all edge thresholds
362        use this to bound allowable values for Edge7 threshold
363    */
364     for(max_thresh = 0; edge = 0; edge < NUMBER_OF_EDGES; edge++)
365     {
366         tmp = ((long)period - (long)end - calib.EdgeOffset[eramp][edge])

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_run.c

DATE 5/23/89  
TIME 4:41:35 pm

PAGE #  
4/201

```

LINE #          SOURCE TEXT
361      / calib.EdgeSlope[eramp][edge],
362      if(max_thresh < tmp)
363      max_thresh = tmp;
364      }
365
366      if(delay < calib.SampleMinDelay[eramp])
367      {
368          ln_queue_message(ERROR_MSG, "delay out of range; delay[6] = %d, SampleMinDelay = %d, max = %d",
369          delay, calib.SampleMinDelay[eramp], (period - end));
370          return(FAILURE);
371      }
372      else
373      {
374          *thresholdptr = 355;
375          for(aramp = 0; aramp < 4; aramp++)
376          {
377              thresh = ((long)delay - calib.SampleOffset[eramp][aramp]
378              - calib.SampleSlope[aramp]*calib.SampleMinThresh[aramp])
379              / calib.Edge7Slope[eramp];
380              if((thresh < *thresholdptr) && (thresh > 0))
381                  *thresholdptr = thresh;
382          }
383          if(*thresholdptr < calib.Edge7MinThresh[0])
384          {
385              ln_queue_message(ERROR_MSG, "requested Edge 7 threshold below minimum");
386              return FAILURE;
387          }
388          if(*thresholdptr > max_thresh)
389          {
390              ln_queue_message(ERROR_MSG, "requested Edge 7 threshold %d is above \
391              maximum for period %d", *thresholdptr, period);
392              return FAILURE;
393          }
394      }
395      return SUCCESS;
396  }
397
398  /* ln_tmg_get_quantization_and_jitter_error(period, quantptr, jitterptr)
399  *
400  * This function returns the quantization step size of delay
401  * with respect to threshold setting, everything in ps.
402  *
403  * Inputs: logical clock period, pointer to error
404  * Outputs: quantization and jitter are assigned based on the edge 0
405  * slope of some ramp, returns SUCCESS or FAILURE
406  */
407  ln_tmg_get_quantization_and_jitter_error(period, quantptr, jitterptr)
408  {
409      u_long period;
410      u_long *quantptr;
411      u_long *jitterptr;
412      {
413          int i;
414          for(i = 0; i < 4; i++)
415          {
416              if(calib.EdgeMaxDelay[i] > period)
417                  break;
418          }
419          if(i == 4) {
420              ln_queue_message(ERROR_MSG, "period too large; period = %d, EdgeMaxDelay[3] = %d",
421              period, calib.EdgeMaxDelay[3]);
422              return(FAILURE);
423          }
424          *quantptr = calib.EdgeSlope[i][0]; /* always 1 threshold worth */
425          *jitterptr = calib.EdgeSlope[i][0]; /* 1 threshold for now */
426      }
427      return(SUCCESS);
428  }
429
430  /* ln_tmg_get_minimum_sample_ramp_setting()
431  *
432  * This function returns minimum threshold settings for the
433  * four sample ramps.
434  *
435  * Inputs: pointers to minimum threshold variables
436  * Outputs: threshold variables are assigned, returns SUCCESS
437  * or FAILURE
438  */
439  ln_tmg_get_minimum_sample_ramp_setting(r0ptr, r1ptr, r2ptr, r3ptr)
440  {
441      u_char *r0ptr;
442      u_char *r1ptr;
443      u_char *r2ptr;
444      u_char *r3ptr;
445      {
446          if(!calib.CalCompleted) {
447              ln_queue_message(ERROR_MSG, "problem was encountered during calibration");
448              return(FAILURE);
449          }
450          *r0ptr = calib.SampleMinThresh[0];
451          *r1ptr = calib.SampleMinThresh[1];
452          *r2ptr = calib.SampleMinThresh[2];
453          *r3ptr = calib.SampleMinThresh[3];
454          return(SUCCESS);
455      }
456  }
457
458  /* ln_tmg_get_sample_ramp_dead_time()
459  *
460  * This function returns the maximum dead time in ps for the SAMPLE
461  * ramps.
462  *
463  * Inputs: logical clock period, mode, pointer to dead time variable
464  * Outputs: dead time assigned, function returns SUCCESS or FAILURE
465  */
466  ln_tmg_get_sample_ramp_dead_time(period, mode, deadptr, jitterptr)
467  {
468      u_long period;
469      u_long mode;
470      u_long *deadptr;
471      u_long *jitterptr;
472      {
473          int aramp;
474          for(aramp = 0; aramp < 4; aramp++)
475          {
476              if(calib.EdgeMaxDelay[aramp] > period)
477                  break;
478          }
479          if(aramp == 4) {
480              ln_queue_message(ERROR_MSG, "period too large; period = %d, EdgeMaxDelay[3] = %d",
481              period, calib.EdgeMaxDelay[3]);
482              return(FAILURE);
483          }
484      }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_run.c

DATE 5/23/89

PAGE #

TIME 4:41:35 pm

5/202

```

LINE # SOURCE TEXT
481 )
482
483 if(!calib.CalCompleted) {
484     lm_queue_message(ERROR_MSG, "problem was encountered during calibration");
485     return(FAILURE);
486 }
487 if(mode == EDGE7SAMPLETRIGGERMODE)
488 {
489     if(eramp == 0) /* don't need to include delay line effects */
490         *deadptr = calib.SampleMinDelay(eramp);
491     else
492         *deadptr = calib.SampleMinDelay(eramp) + 15001; /* add 1 + 3/4 ns */
493 }
494 else
495 {
496     if(eramp == 0)
497         *deadptr = 0; /* ok to place edges at magic chip dead time */
498     else
499         *deadptr = 2 * (calib.SampleMinDelay(eramp) + 15001);
500     /* make sure this is greater than EDGE7 mode dead time */
501 }
502
503 *jitterptr = calib.SampleSlope(3); /* worst case sample jitter */
504
505 return(SUCCESS);
506 }
507
508 /*
509  * lm_tmg_set_frequency()
510  *
511  * This function sets the clock and starts timer using
512  * Counter 1. This timer is used along with lock detect
513  * to decide if clock frequency has stabilized. Counter 1
514  * is loaded with 1750 - 1 and counts at 3.75 MHz, resulting
515  * in a time delay of 1 ns.
516  *
517  * Inputs: none
518  * Outputs: returns SUCCESS or FAILURE
519  */
520 static int tmg_was_changed;
521 lm_tmg_set_frequency(a, k, source)
522     u_char a;
523     u_char k;
524     u_char source;
525 {
526     static u_char tmg_last_a = 0;
527     int reala, realk;
528     int maxtries;
529
530     if((source == EXT0) || (source == EXT1))
531     /* can't set frequency with clock on */
532     if(lm_tmg_clockoff() != SUCCESS)
533         return(FAILURE);
534     tmgptr->clock_select = source;
535     tmg_was_changed = FALSE;
536     if(islow())
537     {
538         lm_queue_message(ERROR_MSG, "External clock is slow");
539         return FAILURE;
540     }
541     else
542         return SUCCESS;
543 }
544
545 reala = 257 - a;
546 realk = 257 - k;
547
548 if((reala < 128) || (realk == 0) || (source > 2)) {
549     lm_queue_message(ERROR_MSG, "illegal value for a,k,source: a = %d, k = %d, source = %d",
550         a, k, source);
551     return(FAILURE);
552 }
553
554 /* can't set frequency with clock on */
555 if(lm_tmg_clockoff() != SUCCESS)
556     return(FAILURE);
557
558 if(a == tmg_last_a) /* pll does not need any lock time */
559 {
560     tmg_was_changed = FALSE;
561     tmgptr->pll_divisor = k;
562     tmgptr->clock_select = source;
563     return(SUCCESS);
564 }
565
566 tmgptr->clock_select = 1; /* don't know what freq, but its square */
567 if(lm_tmg_clockon() != SUCCESS)
568     return(FAILURE);
569
570 tmgptr->ctc_intr_clear = 0;
571 tmgptr->ctc_register[3].value = 0x10; /* setup mode 0 */
572 tmgptr->ctc_register[0].value = zero; /* lab for max count */
573 tmgptr->ctc_register[0].value = zero; /* max for max count */
574
575 /* since GATE0 has no effect on OUT0, OUT0 should be low */
576 /* and furthermore, Counters should not be counting */
577
578 /* at this point, merely verify that Counter0 output is low */
579 maxtries = 10000;
580 while(1)
581 {
582     tmgptr->ctc_register[3].value = LATCH_CNTR_0;
583     if(!((tmgptr->ctc_register[0].value & 0x80))
584         break;
585     if(--maxtries == 0)
586     {
587         lm_queue_message(ERROR_MSG, "CTC failure\n");
588         return FAILURE;
589     }
590 }
591
592 /* turn the clock off to keep the FACS & PELS happy */
593 if(lm_tmg_clockoff() != SUCCESS) /* shut off clock while output true */
594     return(FAILURE);
595
596 tmgptr->ctc_register[3].value = 0x070; /* setup mode 0 */
597 tmgptr->ctc_register[1].value = 0xa5; /* lab of 1750 - 1 */
598 tmgptr->ctc_register[1].value = 0x0e; /* max of 1750 - 1 */
599
600

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_run.c	DATE 5/23/89	PAGE # 6/203
			TIME 4:41:35 pm	
LINE #	SOURCE TEXT			
601	/* timer is now running, using Counter 1 */			
602	tmg_last_a = a;			
603	tmg_a_was_changed = TRUE;			
604	tmgptr->pll_rate = a;			
605	tr->pll_divisor = k;			
606	tmg->clock_select = source;			
607	return(SUCCESS);			
608	}			
609				
610	/*			
611	ls_tmg_check_locked()			
612	/*			
613	This function returns SUCCESS if PLL lock indicator is true			
614	and 1 ms has expired since changing its frequency, as			
615	indicated by Counter 1. Counter 1 is setup by			
616	ls_tmg_set_frequency().			
617	*/			
618	ls_tmg_check_locked()			
619	{			
620	if(!tmg_a_was_changed && tmgptr->pll_locked) /* no need to wait */			
621	return(SUCCESS);			
622	/* when Counter 1 output goes true, desired time delay has passed */			
623	tmgptr->ctc_register[3].value = LATCH_CNTR_1_STATUS;			
624	if(((tmgptr->ctc_register[1].value & 0x0c0) == 0x080) && tmgptr->pll_locked)			
625	return(SUCCESS); /* null flag not set and output true */			
626	else			
627	return(FAILURE);			
628	}			
629				
630				
631				
632				
633	/*			
634	ls_tmg_measure_clock()			
635	/*			
636	This function measures the frequency of either external clock.			
637	*/			
638	Inputs: clock type (external 0 or 1), pointer to clock period in ps			
639	Outputs: clock period is assigned, function returns SUCCESS			
640	or FAILURE.			
641	/*			
642	ls_tmg_measure_clock(clocktype, periodptr)			
643	u_char clocktype;			
644	u_long *periodptr;			
645	{			
646	int i, saveintenable;			
647	u_long least, most, cntrl[3], cntr2[3], start;			
648	if(ls_tmg_clockoff() != SUCCESS)			
649	return(FAILURE);			
650	if((clocktype != EXT0) && (clocktype != EXT1)) {			
651	ls_queue_message(ERROR_MSG, "invalid clock type (%d)",			
652	clocktype);			
653	return(FAILURE);			
654	tmgptr->clock_select = clocktype;			
655	if(ls_tmg_clockon() != SUCCESS)			
656	return(FAILURE);			
657	if(isalow()) {			
658	ls_queue_message(ERROR_MSG, "clock is slow");			
659	return(FAILURE);			
660	}			
661	saveintenable = tmgptr->ctc_intr_enable;			
662	tmgptr->ctc_intr_enable = 0;			
663	tmgptr->ctc_intr_clearl = 0; /* output false */			
664	tmgptr->ctc_register[3].value = CNTR0CM; /* setup 0 */			
665	tmgptr->ctc_register[0].value = 0x00; /* lab of 1000 */			
666	tmgptr->ctc_register[0].value = 0x00; /* mab of 1000 */			
667	tmgptr->ctc_register[3].value = CNTR1CM; /* setup 1,2 */			
668	tmgptr->ctc_register[1].value = zero;			
669	tmgptr->ctc_register[1].value = zero;			
670	tmgptr->ctc_register[3].value = CNTR2CM;			
671	tmgptr->ctc_register[2].value = zero;			
672	tmgptr->ctc_register[2].value = zero;			
673	tmgptr->ctc_intr_clearl = 1; /* let her rip */			
674	start = ls_time();			
675	while(!tmgptr->ctc_intr)			
676	if((ls_time() - start) > 1000)			
677	{			
678	tmgptr->ctc_intr_clearl = 0;			
679	tmgptr->ctc_intr_enable = saveintenable;			
680	ls_queue_message(ERROR_MSG, "timeout, no CTC interrupt");			
681	return(FAILURE);			
682	}			
683	tmgptr->ctc_register[3].value = CNTR1LATCH; /* latch count/status */			
684	for(i = 0; i < 3; i++)			
685	cntrl[i] = tmgptr->ctc_register[1].value & 0x0ff;			
686	for(i = 0; i < 3; i++)			
687	cntr2[i] = tmgptr->ctc_register[2].value & 0x0ff;			
688	if(cntrl[0] & 0x40) /* check null flag */			
689	least = 0;			
690	else			
691	{			
692	least = cntrl[1] + (cntrl[2] << 8);			
693	if(least == 0)			
694	least = 1;			
695	else			
696	least = 0x100011 - least;			
697	}			
698	if(cntr2[0] & 0x40) /* check null flag */			
699	most = 0;			
700	else			
701	{			
702	most = cntr2[1] + (cntr2[2] << 8);			
703	if(most == 0)			
704	most = 1;			
705	else			
706	most = 0x100011 - most;			
707	}			
708	*periodptr = (((most << 16) - least) * 1000) / 60;			
709				
710				
711				
712				
713				
714				
715				
716				
717				
718				
719				
720				

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_run.c	DATE 5/23/89	PAGE # 7/204
LINE #	SOURCE TEXT			
721	tmgptr->ctr_intr_clearl = 0;			
722	tmgptr->ctr_intr_enable = saveintrenable;			
723	return(SUCCESS);			
724	}			
725	}			
726	slow()			
727	{			
728	Register long counter;			
729	u_long start;			
730	}			
731	if(!m_tmg_clockon() != SUCCESS)			
732	return(1); /* same as slow clock */			
733	}			
734	start = m_time();			
735	do			
736	{			
737	if((m_time() - start) > TIMEOUT)			
738	{			
739	m_queue_message(ERROR_MSG, "slow(): error in initialization\n");			
740	return FAILURE;			
741	}			
742	tmgptr->slow_clock_clearl = 0; /* clear it */			
743	tmgptr->slow_clock_clearl = 1; /* unclear it */			
744	} while(tmgptr->slow_clock); /* it set, repeat */			
745	/* wait about 25 msec to see if slow clock */			
746	counter = 49; /* delay = 500 ns * (49 + 1) = 25000 ns */			
747	while(--counter)			
748	{			
749	}			
750	}			
751	return(tmgptr->slow_clock);			
752	}			
753	}			
754	/*			
755	int m_tmg_clockoff(void)			
756	{			
757	/* shuts clock off			
758	/* Input: nothing			
759	/* Output: SUCCESS or FAILURE			
760	*/			
761	int m_tmg_clockoff()			
762	{			
763	u_long start;			
764	tmgptr->clock_enable = 0;			
765	start = m_time();			
766	while(tmgptr->clock_on)			
767	{			
768	if((m_time() - start) > 10)			
769	{			
770	if(tmgptr->clock_select > 2) /* must be external clock */			
771	tmgptr->clock_sync_clearl = 0;			
772	break;			
773	}			
774	if(tmgptr->clock_on)			
775	{			
776	m_queue_message(ERROR_MSG, "can't turn off clock");			
777	return(FAILURE);			
778	}			
779	tmgptr->clock_sync_clearl = 0;			
780	return(SUCCESS);			
781	}			
782	/*			
783	int m_tmg_clockon(void)			
784	{			
785	/* turns clock on			
786	/* Input: nothing			
787	/* Output: SUCCESS or FAILURE			
788	*/			
789	int m_tmg_clockon()			
790	{			
791	u_long start;			
792	if(!m_tmg_clockoff() != SUCCESS)			
793	return FAILURE;			
794	}			
795	tmgptr->clock_sync_clearl = 1;			
796	}			
797	tmgptr->clock_enable = 1;			
798	start = m_time();			
799	while(!tmgptr->clock_on)			
800	{			
801	if((m_time() - start) > 10) {			
802	m_queue_message(ERROR_MSG, "can't turn on clock");			
803	return(FAILURE);			
804	}			
805	}			
806	return(SUCCESS);			
807	}			
808	/*			
809	m_tmg_set_falling_sample(count)			
810	{			
811	/*			
812	This function sets the sample width counter with the			
813	specified count after checking it against bounds.			
814	/*			
815	Inputs: count (number of clock cycles - 1 for width)			
816	/*			
817	Outputs: function returns SUCCESS or FAILURE			
818	*/			
819	m_tmg_set_falling_sample(count)			
820	{			
821	if(count > 255) {			
822	m_queue_message(ERROR_MSG, "value too large (td)");			
823	return(FAILURE);			
824	}			
825	else			
826	{			
827	tmgptr->sample_width = count;			
828	return(SUCCESS);			
829	}			
830	}			
831	/*			
832	m_tmg_set_rising_sample(range, threshold)			
833	{			
834	/*			
835	This function sets the comparator threshold used to			
836	determine rising edge of SAMPLE. Threshold is checked			
837	against bounds.			
838	/*			
839	Inputs: sample ramp and threshold value			
840	/*			
	Outputs: function returns SUCCESS or FAILURE			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_run.c

DATE

5/23/89

PAGE #

TIME

4:41:35 pm

8/205

```

LINE #          SOURCE TEXT
841  /*
842  lm_tmg_set_rising_sample(range, threshold)
843  u_long range,
844  u_long threshold;
845  {
846  if((range > 3) || (threshold > 255) ||
847  (threshold < calib.SampleMinThresh(range))) {
848  lm_queue_message(ERROR_MSG, "range too large (%d) OR threshold out of range (%d)",
849  range, threshold);
850  return(FAILURE);
851  }
852  else
853  {
854  tmgptr->sample_delay = threshold;
855  tmgptr->sample_delay_range = range;
856  return(SUCCESS);
857  }
858  }
859  /*
860  lm_tmg_set_edge_settings(period, thresholds)
861  /*
862  * This function sets the comparator thresholds used
863  * for the six timing edges and the sample ramp trigger.
864  * Thresholds are checked against bounds.
865  * Inputs: logical clock period in ps, and threshold array
866  * Outputs: function returns SUCCESS or FAILURE
867  */
868  lm_tmg_set_edge_settings(period, thresholds)
869  u_long period,
870  u_long thresholds[];
871  {
872  int ramp;
873  for(ramp = 0; ramp < 4; ramp++)
874  if(calib.EdgeMaxDelay[ramp] > period)
875  return lm_tmg_set_ramp_edge_settings(ramp, thresholds);
876  lm_queue_message(ERROR_MSG, "period too large, period = %d, EdgeMaxDelay[3] = %d",
877  period, calib.EdgeMaxDelay[3]);
878  return(FAILURE);
879  }
880  lm_tmg_set_ramp_edge_settings(ramp, thresholds)
881  register u_long ramp;
882  u_long thresholds[];
883  {
884  register u_long edge;
885  for(edge = 0; edge < 6; edge++)
886  if((thresholds[edge] > 255) ||
887  (thresholds[edge] < calib.EdgeMinThresh(ramp)[edge])) {
888  lm_queue_message(ERROR_MSG, "threshold for edge %d is out of range (%d)",
889  edge, thresholds[edge]);
890  return(FAILURE);
891  }
892  else
893  tmgptr->edge_delay[edge].delay = thresholds[edge];
894  #ifdef possible_bug
895  if((thresholds[6] > 255) || (thresholds[6] < calib.EdgeMinThresh(ramp)[6]))
896  #else possible_bug
897  if((thresholds[6] > 255) || (thresholds[6] < calib.Edge7MinThresh(ramp)))
898  #endif possible_bug
899  return(FAILURE);
900  else
901  tmgptr->sample_trigger_threshold = thresholds[6]; /* edge "7" */
902  tmgptr->edge_delay_range = ramp;
903  return(SUCCESS);
904  }
905  /*
906  int lm_tmg_play(timeout)
907  /*
908  * Initiates pattern presentation
909  * Inputs: none
910  * Returns: SUCCESS or FAILURE
911  */
912  int lm_tmg_play(timeout)
913  u_long timeout;
914  {
915  if(lm_tmg_initiate_play() != SUCCESS)
916  return(FAILURE);
917  return(lm_tmg_complete_play(timeout));
918  }
919  /*
920  int lm_tmg_initiate_play(void)
921  /*
922  * This routine turns clocks on and starts a presentation.
923  * Inputs: none
924  * Outputs: function returns SUCCESS or FAILURE
925  */
926  int lm_tmg_initiate_play()
927  {
928  if(tmgptr->lase_intr) /* is error on bus? */
929  if(tmgptr->lase_intr) /* really? */
930  {
931  lm_queue_message(ERROR_MSG, "lm_tmg_initiate_play: error on bus");
932  return(FAILURE);
933  }
934  if(tmgptr->backplane_mode) /* is PLAY mode? */
935  {
936  lm_queue_message(ERROR_MSG, "lm_tmg_initiate_play: already in PLAY mode");
937  return(FAILURE);
938  }
939  if(lm_tmg_clockoff() != SUCCESS)
940  return(FAILURE);
941  if(lm_tmg_clockon() != SUCCESS)
942  return(FAILURE);
943  play_completed_flag = 0;
944  post_end_of_play();
945  tmgptr->pattern_intr_enable = 0; /* just in case */
946  tmgptr->pattern_intr_enable = 1; /* enable EOP interrupt */
947  tmgptr->start_pattern_play = 1; /* start presentation */
948  return(SUCCESS);
949  }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_run.c

DATE	5/23/89	PAGE #
TIME	4:41:35 pm	9/206

```

LINE #          SOURCE TEXT
961  }
962
963
964  /*      int lm_tag_complete_play(u_long timeout)
965  *
966  *      This function waits for a presentation to complete by
967  *      waiting for the EOP interrupt from the Timing Generator.
968  *
969  *      Inputs: timeout is ms
970  *      Outputs: function returns SUCCESS or FAILURE
971  */
972  int lm_tag_complete_play(timeout)
973  u_long timeout;
974  {
975      u_long start;
976
977      #ifdef DIAGS
978      #ifdef MODELER
979      int err;
980
981      while (! play_completed_flag)
982      {
983          ac_send( play_semaphore, lm_number_of_ticks( timeout ), &err );
984          if( err == VRTX_TIMEOUT )
985          {
986              lm_tag_abort_play();
987              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
988              post_end_of_play=0;
989              return( FAILURE );
990          }
991          if( err != VRTX_OK )
992          {
993              post_end_of_play=0;
994              printf("Error in lm_tag_complete_play(), error code %d\n",err);
995              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
996              return(FAILURE);
997          }
998      }
999      post_end_of_play=0;
1000      tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1001      return(SUCCESS);
1002      #endif
1003      #else
1004      start = lm_time();
1005      while(!TMC_INT)
1006      {
1007          if((lm_time() - start) > timeout)      /* wait for interrupt */
1008          {
1009              lm_tag_abort_play();
1010              tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1011              return(FAILURE);
1012          }
1013          tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1014          return(SUCCESS);
1015      }
1016      #endif
1017
1018  /*      int lm_tag_abort_play()
1019  *
1020  *      INPUT: none
1021  *      OUTPUT: returns SUCCESS or FAILURE
1022  *      DESCRIPTION: Aborts pattern play by asserting and then removing
1023  *      the error line.
1024  */
1025  int
1026  lm_tag_abort_play()
1027  {
1028      u_long start;
1029      int stuck, returncode;
1030      int save_lane_intr_enable;
1031
1032      save_lane_intr_enable = tmgptr->lane_intr_enable;
1033
1034      stuck = FALSE;
1035      returncode = SUCCESS;
1036
1037      if(!tmgptr->backplane_mode)      /* if ACCESS mode */
1038      {
1039          tmgptr->pattern_intr_enable = 0;      /* clear EOP interrupt */
1040          if(lm_tag_clockoff() != SUCCESS)
1041          {
1042              lm_queue_message(ERROR_MSG, "lm_tag_complete_play: clockoff returned error");
1043              return(FAILURE);
1044          }
1045      }
1046      else
1047      {
1048          disable_mod_err();
1049          if(tmgptr->lane_enable)
1050          {
1051              if(tmgptr->lane_enable & ~tmgptr->lane_intr)
1052              {
1053                  start = lm_time();
1054                  tmgptr->lane_intr_enable = 0;
1055                  tmgptr->backplane_error = 1;      /* Assert error line */
1056                  while(tmgptr->clock_on)      /* Wait for clock to stop */
1057                  {
1058                      if((lm_time() - start) > 10)      /* 10 ms timeout */
1059                      {
1060                          stuck = TRUE;
1061                          lm_queue_me_("%(ERROR_MSG, "lm_tag_abort_play: PAC did not respond to ERROR");
1062                          break;
1063                      }
1064                      tmgptr->backplane_error = 0;      /* Deassert error line */
1065                  }
1066              }
1067          }
1068          if(stuck)
1069          {
1070              tmgptr->abort_pattern_play = 1;      /* last resort */
1071              start = lm_time();
1072              while(tmgptr->clock_on)
1073              {
1074                  if((lm_time() - start) > 10)
1075                  {
1076                      returncode = FAILURE;
1077                      lm_queue_message(ERROR_MSG, "lm_tag_abort_play: TMC state machine did not quit");
1078                      lm_queue_message(ERROR_MSG, "mode = %d",
1079                                      tmgptr->backplane_mode);
1080                      lm_queue_message(ERROR_MSG, "clock_on = %d",
1081                                      tmgptr->clock_on);
1082                      lm_queue_message(ERROR_MSG, "errors = %x",
1083                                      tmgptr->lane_intr);
1084                  }
1085              }
1086          }
1087      }
1088  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_run.c

DATE	5/23/89	PAGE #
TIME	4:41:35 pm	10/207

LINE #	SOURCE TEXT
1081	}
1082	}
1083	
1084	/* Finish error signal out of pipeline */
1085	if(lm_tmg_clockoff() != SUCCESS) /* Turn off clock */
1086	returncode = FAILURE;
1087	if(lm_tmg_clockon() != SUCCESS) /* Turn on clock */
1088	returncode = FAILURE;
1089	if(lm_tmg_clockoff() != SUCCESS) /* Turn off clock */
1090	returncode = FAILURE;
1091	
1092	tmgptr->lase_intr_enable = save_lase_intr_enable;
1093	
1094	enable_mod_err();
1095	return(returncode);
1096	}
1097	
1098	
1099	/*
1100	lm_tmg_set_sample_trigger_mode(mode)
1101	*/
1102	/* Selects either normal or early sample trigger mode.
1103	*/
1104	/* Input: mode
1105	/* Output: Returns SUCCESS or FAILURE if mode is not 0 or 1
1106	*/
1107	lm_tmg_set_sample_trigger_mode(mode)
1108	u_long mode;
1109	{
1110	if(mode > 1)
1111	{
1112	lm_queue_message(ERROR_MSG, "lm_tmg_set_sample_trigger_mode: invalid mode");
1113	return FAILURE;
1114	}
1115	else
1116	{
1117	tmgptr->sample_mode = mode;
1118	return SUCCESS;
1119	}
1120	}
1121	
1122	

Copyright 1989 Logic Modeling Systems		HEADER FILE diags/tmg_run.h	DATE 5/23/89 TIME 4:41:37 pm	PAGE # 1/208
LINE #	HEADER TEXT			
1	/* SCCS ID: tmg_run.h rev 1.1, 4/24/89 at 07:51:00 */			
2				
3	#define N_MIN 128			
4	#define N_MAX 256			
5	#define T_REF 8533333.3 /* reference period in ps */			
6	#define MAX_PERIOD 6667746 /* 1 / 150 KHz in ps */			
7				
8	#define MIN_PERIOD 40000.0 /* actually our approx of 150 MHz */			
9	#define DEAD_TIME 15000 /* 1 / 25 MHz in ps */			
10	#define LATCH_CNTR_0 /* guess in ps */			
11	#define LATCH_CNTR_0 0x0e2 /* latch count in Counter 0 */			
12	#endif			
13	#define LATCH_CNTR_1			
14	#define LATCH_CNTR_1 0x0c4 /* latch count in Counter 1 */			
15	#endif			
16	#define LATCH_CNTR_1_STATUS 0xe4 /* latch status only */			
17				
18	#define EXT0 3 /* register value to select ext clock 0 */			
19	#define EXT1 4 /* register value to select ext clock 1 */			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM diags/tmg_util.c	DATE 5/23/89	PAGE # 1/209
LINE #	SOURCE TEXT			
1	/* SCOS ID: tmg_util.c.rev:3.1, 4/26/89 at 07:51:03 */			
2	/*			
3	* tmg_util.c			
4	*			
5	* A collection of small utility routines for the Timing			
6	* Generator Diagnostics and Calibration			
7	*			
8	*			
9	*			
10	*/			
11	#include "common.h"			
12	#include "moduler_extn.h"			
13	#include "mod_def.h"			
14	#include "tmg.h"			
15	#include "tmg_def.h"			
16	#include "vtr.h"			
17	#include "tmg_extn.h"			
18	#include "diag.h"			
19	#include "diag_setjmp.h"			
20	#include "instr.h"			
21	static int zero = 0;			
22	/*			
23	* u_long tmg_measure_freq(void)			
24	* This routine measures clock frequency of the selected clock.			
25	* It counts how many unknown clocks occur within 10,000 clocks			
26	* of 3.75 MHz. If the unknown frequency is as low as 150 KHz,			
27	* there can be as many as 250,000 reference clocks, thus the			
28	* need for a 32 bit count of reference clocks.			
29	* Inputs: none			
30	* Returns: number of clocks (3.75 MHz, 256.7 ns period)			
31	*/			
32	u_long tmg_measure_freq(soclockptr)			
33	u_long *soclockptr;			
34	{			
35	int i;			
36	u_long least, most, cnt1[3], cnt2[3], start;			
37	if(tmg_clockoff() != SUCCESS)			
38	{			
39	lm_error("Measure Freq: could not turn clock off.\n");			
40	return(FAILURE);			
41	}			
42	if(tmg_clockon() != SUCCESS)			
43	{			
44	lm_error("Measure Freq: could not turn clock on.\n");			
45	lm_message("Please check the external clock connection\n");			
46	return(FAILURE);			
47	}			
48	if(tmg_islow())			
49	{			
50	lm_warning("The frequency is too slow.\n");			
51	lm_warning("==> Please check the external clock connection.\n");			
52	lm_warning("==> Allowed range is 150 KHz to 25 MHz, 0-4 V p-p.\n");			
53	}			
54	/* output false */			
55	tmgptr->ctc_intr_clearl = 0;			
56	tmgptr->ctc_register[3].value = CNTR0CN, /* setup 0 */			
57	tmgptr->ctc_register[0].value = 0x10, /* 1st of 10000 */			
58	tmgptr->ctc_register[0].value = 0x17, /* 2nd of 10000 */			
59	tmgptr->ctc_register[3].value = CNTR1CN, /* setup 1,2 */			
60	tmgptr->ctc_register[1].value = zero;			
61	tmgptr->ctc_register[1].value = zero;			
62	tmgptr->ctc_register[3].value = CNTR2CN,			
63	tmgptr->ctc_register[2].value = zero;			
64	tmgptr->ctc_register[2].value = zero;			
65	tmgptr->ctc_register[2].value = zero;			
66	tmgptr->ctc_intr_clearl = 1, /* let bar trip */			
67	start = lm_time();			
68	while(!tmgptr->ctc_intr) /* wait for finish */			
69	{			
70	if((lm_time() - start) > 5000)			
71	{			
72	lm_error("Measure Freq: timeout error waiting for CTC.\n");			
73	goto failure;			
74	}			
75	tmgptr->ctc_register[3].value = CNTR2LATCH, /* latch count/status */			
76	for(i = 0; i < 3; i++)			
77	cnt1[i] = tmgptr->ctc_register[i].value & 0x0ff;			
78	for(i = 0; i < 3; i++)			
79	cnt2[i] = tmgptr->ctc_register[i].value & 0x0ff;			
80	if(cnt1[0] & 0x40) /* check null flag */			
81	{			
82	least = 0;			
83	{			
84	least = cnt1[1] + (cnt1[2] << 8);			
85	if(least == 0)			
86	least = 1;			
87	else			
88	least = 0x100011 - least;			
89	}			
90	if(cnt2[0] & 0x40) /* check null flag */			
91	{			
92	most = 0;			
93	{			
94	most = cnt2[1] + (cnt2[2] << 8);			
95	if(most == 0)			
96	most = 1;			
97	else			
98	most = 0x100011 - most;			
99	}			
100	*soclockptr = (most << 16) + least;			
101	if(tmg_clockoff() != SUCCESS)			
102	{			
103	lm_error("Measure Freq: could not turn clock off before successful exit.\n");			
104	return(FAILURE);			
105	}			
106	return(SUCCESS);			
107	failure:			
108	if(tmg_clockoff() != SUCCESS)			
109	{			
110	lm_error("Measure Freq: could not turn clock off before successful exit.\n");			
111	return(FAILURE);			
112	}			
113	return(SUCCESS);			
114	failure:			
115	if(tmg_clockoff() != SUCCESS)			
116	{			
117	lm_error("Measure Freq: could not turn clock off before successful exit.\n");			
118	return(FAILURE);			
119	}			
120	return(SUCCESS);			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_util.c

DATE 5/23/89  
TIME 4:41:37 pm

PAGE #  
2/210

```

121  ln_error("Measure freq: could not turn clock off before aborted exit.\n");
122  return(FAILURE);
123  }
124  return FAILURE;
125  }
126
127  tmg_measure_period()
128  {
129      u_long sum_of_clocks;
130
131      /* Measure the clock speed */
132      if (tmg_measure_freq(sum_of_clocks) != SUCCESS)
133      {
134          (void)ln_error("Clock frequency measurement failed.\n");
135          return 0;
136      }
137      /* return pattern clock period in picoseconds */
138      return ((sum_of_clocks * 101) / 61);
139  }
140
141
142  /* int tmg_isallow(void)
143   * Checks if clock is slow or not
144   * Inputs: none
145   * Output: 1 if slow, 0 if not
146   */
147  int tmg_isallow()
148  {
149      tmgptr->slow_clock_clear1 = 0;
150      tmgptr->slow_clock_clear1 = 1;
151      ln_delay(10);
152      return(tmgptr->slow_clock);
153  }
154
155  /* int tmg_toggle(int edge, int reccount, int mode)
156   * This routine checks to see if the specified edge's
157   * calibration flip flop can toggle with the current
158   * setup (delay line settings, threshold, clock rate, etc).
159   * In particular, the proper test mode, if any, must be setup.
160   * Inputs: the edge number to toggle 0..3
161   *         reccount, and a mode bit
162   * Returns: number of toggles, or -1 on error
163   */
164
165  int tmg_toggle(edge, reccount, mode)
166  {
167      int edge, reccount, mode;
168
169      int current, mask, previous, timestoggled;
170
171      if((edge < 0) || (edge > 5))
172      {
173          sys_out("tmg_toggle: invalid edge specified\n");
174          return(-1);
175      }
176
177      if(tmg_calclear() != SUCCESS)
178      {
179          sys_out("tmg_toggle: cal flip flops did not clear\n");
180          return(-1);
181      }
182
183      timestoggled = 0;
184      previous = 0;
185      mask = 1 << edge;
186
187      if(mode == TOGGLEMODE)
188      {
189          while(reccount-- > 0)
190          {
191              if(tmg_play(TIMEOUT) != SUCCESS)
192              {
193                  sys_out("tmg_toggle: tmg_play(TIMEOUT) returned error\n");
194                  return(-1);
195              }
196              if((current = tmgptr->edge_cal & mask) != previous)
197              {
198                  previous = current;
199                  timestoggled++;
200              }
201              else
202              {
203                  break;
204              }
205          }
206      }
207      else if(mode == NOTOGGLEMODE)
208      {
209          while(reccount-- > 0)
210          {
211              if(tmg_play(TIMEOUT) != SUCCESS)
212              {
213                  sys_out("tmg_toggle: tmg_play(TIMEOUT) returned error\n");
214                  return(-1);
215              }
216              if((current = tmgptr->edge_cal & mask) != previous)
217              {
218                  timestoggled++;
219                  break;
220              }
221          }
222      }
223      else
224      {
225          return(-1);
226      }
227
228      return(timestoggled);
229  }
230
231  /* int tmg_toggle_all(count, no_toggles)
232   * This function checks all edge calibration flip flops for toggling.
233   * This should be used when searching through clock periods to find
234   * points along ramps.
235   * Inputs: count, array of toggles for each edge
236   * Outputs: array of toggles is assigned, function returns SUCCESS
237   *          or FAILURE
238   */
239  int tmg_toggle_all(count, no_toggles)
240  {
241      int count, no_toggles[];

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

3/211

```

LINE # SOURCE TEXT
241 u_char data;
242 int i;
243
244 if(two_effclear() != SUCCESS)
245 {
246     lm_error("tmg_toggle_all: cal flip flops did not clear\n");
247     return(FAILURE);
248 }
249
250 for(i = 0; i < 6; i++)
251     no_toggles[i] = 0;
252 data = 0;
253
254 while(count--)
255 {
256     if(tmg_play(TIMEOUT) != SUCCESS)
257     {
258         lm_error("tmg_toggle_all: tmg_play(TIMEOUT) returned error\n");
259         return(FAILURE);
260     }
261     data = tmgptr->edge_cal ^ data;
262     for(i = 0; i < 6; i++)
263         no_toggles[i] += (data >> i) & 1;
264     return(SUCCESS);
265 }
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360

```

This function initializes the Timing Generator to default settings.  
 Inputs: none  
 Outputs: function returns SUCCESS or FAILURE

```

int tmg_init()
{
    if(Reset)
        return(SUCCESS);

    calib.CalCompleted = 0;
    return diag_tmg_reset(TRUE);
}

int tmg_walk1(p)
unsigned char *p;
{
    int i, j, returncode;
    unsigned char tmp;

    returncode = SUCCESS;
    for(i = 1, j = 0; j < 8; j++, i <= 1)
    {
        *p = i;
        if((tmp = *p) != i)
        {
            lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp);
            returncode = FAILURE;
        }
    }

    return(returncode);
}

int tmg_walk0(p)
unsigned char *p;
{
    unsigned int j, returncode;
    unsigned char i, tmp;

    returncode = SUCCESS;
    for(i = 1, j = 0; j < 8; i <= 1, j++)
    {
        *p = ~i;
        if((tmp = *p) != ~i)
        {
            lm_error("walking0: address %08x wrote %02x read %02x\n", p, i, tmp);
            returncode = FAILURE;
        }
    }

    return(returncode);
}

int tmg_req0walk1(p)
unsigned char *p;
{
    int i, j, returncode;
    unsigned char tmp;

    returncode = SUCCESS;
    for(i = 1, j = 0; j < 8; j++, i <= 1)
    {
        if(j < 2)
        {
            *p = 0x10; /* got to remove ORSHIFT */
            *p = i | 0x10;
            if((tmp = *p) != (i | 0x10))
            {
                lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp);
                returncode = FAILURE;
            }
        }
        else
        {
            *p = i;
            if((tmp = *p) != i)
            {
                lm_error("walking1: address %08x wrote %02x read %02x\n", p, i, tmp);
                returncode = FAILURE;
            }
        }
    }

    return(returncode);
}

int tmg_req0walk0(p)

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_util.c

DATE 5/23/89

PAGE #

TIME 4:41:37 pm

4/212

```

LINE # SOURCE TEXT
361 unsigned char *p;
362 {
363     int j, returncode;
364     unsigned char i, tmp;
365
366     returncode = SUCCESS;
367     *p = 0x0ff; /* remove ORESET */
368     for(i = 1, j = 0; j < 8; i <= 1, j++)
369     {
370         *p = ~i;
371         if(j != 4)
372         {
373             if((tmp = *p) != ~i)
374             {
375                 lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
376                 returncode = FAILURE;
377             }
378         }
379         else
380         {
381             if((tmp = *p) != (~i & 0x0fc))
382             {
383                 lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
384                 returncode = FAILURE;
385             }
386             *p = 0x0ff; /* remove ORESET */
387         }
388     }
389     return(returncode);
390 }
391
392 int tmg_req2walk0(p)
393 unsigned char *p;
394 {
395     int j, returncode;
396     unsigned char i, tmp;
397
398     returncode = SUCCESS;
399     for(i = 0x7e, j = 0; j < 8; j++)
400     {
401         *p = i;
402         if((tmp = *p) != i)
403         {
404             lm_error("walking0: address 408x wrote 402x read 402x\n", p, i, tmp);
405             returncode = FAILURE;
406         }
407         i = ((i < 1) | 0x01) & 0x7f; /* have to keep values
408                                     /* within proper range */
409     }
410     return(returncode);
411 }
412
413 /*
414  * int checkrate(int, int, unsigned long, unsigned long *)
415  * This routine checks rate of the clock using the 8254
416  * Input: value of N
417  * value of output division (2,4,6,...,512,516,520,...,1024)
418  * number of clocks expected
419  * Returns: discrepancy is assigned, function returns SUCCESS
420  * or FAILURE
421  */
422 int tmg_checkrate(n,k,nsclcks, discrepascptr)
423 int n, k;
424 unsigned long nsclcks;
425 int *discrepascptr;
426 {
427     int i, ctrl[3], ctrl2[3], returncode;
428     unsigned long actual, most, least, start;
429
430     if(tmg_clockoff() != SUCCESS) /* shut off clock */
431     {
432         lm_error("Checkrate: cannot turn clock off.\n");
433         return(FAILURE);
434     }
435     returncode = SUCCESS;
436     if((n < 128) || (n > 256))
437     {
438         lm_warning("Checkrate: 4d invalid value of N passed.\n", n);
439         return(FAILURE);
440     }
441     if((k < 2) || (k > 1024) || (k & 1) || ((k > 512) && (k%4 != 0)))
442     {
443         lm_warning("Checkrate: 4d invalid value of K passed.\n", k);
444         return(FAILURE);
445     }
446     tmgptr->pll_rate = (256 - n) + 1;
447     if(k == 2)
448     {
449         tmgptr->pll_divisor = 255; /* have to for fo/2 */
450         tmgptr->clock_select = 0; /* select fo/2 */
451     }
452     else if(k <= 512)
453     {
454         tmgptr->pll_divisor = (256 - (k >> 1)) + 1; /* fo/2k */
455         tmgptr->clock_select = 1; /* select fo/2k */
456     }
457     else
458     {
459         tmgptr->pll_divisor = (256 - (k >> 2)) + 1; /* fo/4k */
460         tmgptr->clock_select = 2; /* select fo/4k */
461     }
462     lm_delay(6); /* wait a long time for lock */
463     /* now that PLL is ready, do the thing with the 8254 */
464     tmgptr->ctcr_register[3].value = CTRL1CH; /* setup 1,2 first */
465     tmgptr->ctcr_register[1].value = zero;
466     tmgptr->ctcr_register[1].value = zero;
467     tmgptr->ctcr_register[3].value = CTRL2CH;
468     tmgptr->ctcr_register[2].value = zero;
469     tmgptr->ctcr_register[2].value = zero;
470 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
diags/tmg\_util.c

DATE 5/23/89  
TIME 4:41:37 pm

PAGE #  
5/213

```

LINE # SOURCE TEXT
481 tmgptr->ctc_register[3].value = CTR12LATCH, /* latch count */
482 tmgptr->ctc_register[0].value = 0x00, /* msb of 1000 */
483 tmgptr->ctc_register[0].value = 0x01, /* msb of 1000 */
484
485 tmgptr->ctc_intr_clear1 = 0; /* make sure things are off */
486 if(tmg_clocks() != SUCCESS)
487 {
488     lm_error("Checkrate: cannot turn clock on.\n");
489     return(FAILURE);
490 }
491 tmgptr->ctc_intr_clear1 = 1; /* let her rip */
492
493 start = lm_time();
494 while(!tmgptr->ctc_intr) /* wait for finish */
495     if((lm_time() - start) > 150) /* longest time is 137 ms */
496     {
497         lm_error("Checkrate: timer timed out, a = td, k = td, noclocks = td\n", a, k, noclocks);
498         return(FAILURE);
499     }
500
501 tmgptr->ctc_register[3].value = CTR12LATCH, /* latch count */
502 for(i = 0; i < 3; i++) /* and status */
503     ctrl1[i] = tmgptr->ctc_register[1].value & 0x0f,
504     for(i = 0; i < 3; i++)
505         ctrl2[i] = tmgptr->ctc_register[2].value & 0x0f,
506
507 /* compute actual number of clocks that occurred */
508 if(ctrl1[0] & 0x40) /* check null flag */
509     least = 0;
510 else
511 {
512     least = ctrl1[1] + (ctrl1[2] << 8);
513     if(least == 0)
514         least = 1;
515     else
516         least = 0x100011 - (unsigned long)least;
517 }
518
519 if(ctrl2[0] & 0x40) /* check null flag */
520     most = 0;
521 else
522 {
523     most = ctrl2[1] + (ctrl2[2] << 8);
524     if(most == 0)
525         most = 1;
526     else
527         most = 0x100011 - (unsigned long)most;
528 }
529
530 actual = (most << 16) + least;
531 *discrepancyptr = actual - noclocks;
532 return(returncode);
533 }
534
535 /*
536 * tmg_display_error()
537 * This function displays the error latches for the various
538 * lanes.
539 * Inputs: lanes to display, bit 0 = lane A, bit 3 = lane D
540 * Returns: lm_error returncode
541 */
542 int tmg_display_error(lanes)
543 int lanes;
544 {
545     int i;
546     if(!(lanes & 0x0f)) /* no lanes specified */
547     {
548         return(lm_error("tmg_display_error: no lanes specified\n"));
549     }
550     if(!tmgptr->tmg_intr)
551     {
552         return(lm_error("tmg_display_error: no error condition present\n"));
553     }
554
555     lm_message("\nTMC Error Register Display:\n");
556     lm_message("\nlane(a)\t");
557     for(i = 1; i != 0x10; i <= 1)
558     {
559         if(i & lanes)
560             switch(i)
561             {
562                 case 1: lm_message("\tA"); break;
563                 case 2: lm_message("\tB"); break;
564                 case 4: lm_message("\tC"); break;
565                 case 8: lm_message("\tD"); break;
566                 default: break;
567             }
568         lm_message("\nmpel_ctrl(2..0)");
569         for(i = 1; i != 0x10; i <= 1)
570             if(i & lanes)
571                 switch(i)
572                 {
573                     case 1: lm_message("\ttd", tmgptr->lane_a_pel_control); break;
574                     case 2: lm_message("\ttd", tmgptr->lane_b_pel_control); break;
575                     case 4: lm_message("\ttd", tmgptr->lane_c_pel_control); break;
576                     case 8: lm_message("\ttd", tmgptr->lane_d_pel_control); break;
577                     default: break;
578                 }
579             lm_message("\nData Valid");
580             for(i = 1; i != 0x10; i <= 1)
581                 if(i & lanes)
582                     switch(i)
583                     {
584                         case 1: lm_message("\ttd", tmgptr->lane_a_data_valid); break;
585                         case 2: lm_message("\ttd", tmgptr->lane_b_data_valid); break;
586                         case 4: lm_message("\ttd", tmgptr->lane_c_data_valid); break;
587                         case 8: lm_message("\ttd", tmgptr->lane_d_data_valid); break;
588                         default: break;
589                     }
590             lm_message("\n\n");
591             return(lm_error("Failures in table.\n"));
592         }
593     }
594     /*
595     * tmg_verify_pel_ctrl_error()
596     */

```



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

6/214

```

LINE #          SOURCE TEXT
601  *
602  * This function verifies that the specified bit in the specified lane
603  * disagrees with all other specified lanes.
604  *
605  * Inputs: lanes that participated in the presentation, lane with
606  *         the error, and its bit position
607  * Outputs: function returns SUCCESS if error is verified, else FAILURE
608  */
609 int tmg_verify_pel_ctrl_error(lanes, badlane, bitpos)
610 int lanes, badlane, bitpos;
611 {
612     int a, b, i, mask, returncode;
613     char lane;
614
615     if(! (lanes & 0x0f))
616     {
617         lm_message("tmg_verify_pel_ctrl: no lanes specified\n");
618         return(SUCCESS);
619     }
620     if(lanes == badlane)
621     {
622         lm_message("tmg_verify_pel_ctrl: only one lane... no discrepancies\n");
623         return(SUCCESS);
624     }
625     if((badlane < 0) || (badlane > 0x0f))
626     {
627         lm_error("tmg_verify_pel_ctrl: invalid bad lane specification\n");
628         return(FAILURE);
629     }
630     if((bitpos > 2) || (bitpos < 0))
631     {
632         lm_error("tmg_verify_pel_ctrl: invalid bit position specified\n");
633         return(FAILURE);
634     }
635     mask = 1 << bitpos;
636
637     returncode = SUCCESS;
638     switch(badlane)
639     {
640         case 1: b = tmgptr->lane_a_pel_control; break;
641         case 2: b = tmgptr->lane_b_pel_control; break;
642         case 4: b = tmgptr->lane_c_pel_control; break;
643         case 8: b = tmgptr->lane_d_pel_control; break;
644         default: break;
645     }
646     for(i = 1; i != 0x10; i <= 1)
647     {
648         if(i == badlane)
649             continue;
650         if(i & lanes)
651         {
652             switch(i)
653             {
654                 case 1: a = tmgptr->lane_a_pel_control; lane = 'A'; break;
655                 case 2: a = tmgptr->lane_b_pel_control; lane = 'B'; break;
656                 case 4: a = tmgptr->lane_c_pel_control; lane = 'C'; break;
657                 case 8: a = tmgptr->lane_d_pel_control; lane = 'D'; break;
658                 default: break;
659             }
660             if(!((a ^ b) & mask))
661             {
662                 lm_error("tmg_verify_pel_ctrl: bad data from lane %c, bit %d\n", lane, bitpos);
663                 returncode = FAILURE;
664             }
665         }
666     }
667     return(returncode);
668 }
669
670 int tmg_verify_data_valid_error()
671 *
672 * This function verifies that the specified data valid bit in the
673 * specified lane disagrees with all other specified lanes.
674 *
675 * Inputs: lanes participating in the presentation, lane to check
676 * Outputs: function returns SUCCESS if error is verified, else FAILURE
677 */
678 int tmg_verify_data_valid_error(lanes, badlane)
679 int lanes, badlane;
680 {
681     int a, b, i, returncode;
682     char lane;
683
684     returncode = SUCCESS;
685
686     if(! (lanes & 0x0f))
687     {
688         lm_message("tmg_verify_data_valid_error: no lanes specified\n");
689         return(SUCCESS);
690     }
691     if(lanes == badlane)
692     {
693         lm_message("tmg_verify_data_valid_error: only one lane, no discrepancy\n");
694         return(SUCCESS);
695     }
696     if((badlane < 0) || (badlane > 0x0f))
697     {
698         lm_error("tmg_verify_data_valid_error: invalid bad lane specified\n");
699         return(FAILURE);
700     }
701     switch(badlane)
702     {
703         case 1: b = tmgptr->lane_a_data_valid; break;
704         case 2: b = tmgptr->lane_b_data_valid; break;
705         case 4: b = tmgptr->lane_c_data_valid; break;
706         case 8: b = tmgptr->lane_d_data_valid; break;
707         default: break;
708     }
709     for(i = 1; i != 0x10; i <= 1)
710     {
711         if(i == badlane)
712             continue;
713         if(i & lanes)
714         {
715             switch(i)
716             {
717                 case 1: a = tmgptr->lane_a_data_valid; lane = 'A'; break;
718                 case 2: a = tmgptr->lane_b_data_valid; lane = 'B'; break;
719                 case 4: a = tmgptr->lane_c_data_valid; lane = 'C'; break;
720                 case 8: a = tmgptr->lane_d_data_valid; lane = 'D'; break;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

7/215

```

721 #
722     default: break;
723 }
724 if((a - b))
725 {
726     lm_error("tmg_verify_data_valid_error: no disagreement between bad lane and lane %c\n", lane);
727     returncode = FAILURE;
728 }
729 }
730 return(returncode);
731 }
732
733 /*
734  * int tmg_clear_error(void)
735  *
736  * This function clears the TMG detected backplane error.
737  * It returns SUCCESS if successful, else ERROR.
738  */
739
740 int tmg_clear_error()
741 {
742     if(!tmgptr->tmg_intr) /* no interrupt to clear */
743         return(SUCCESS);
744     tmgptr->tmg_intr_clearl = 0;
745     tmgptr->tmg_intr_clearh = 1;
746     if(!tmgptr->clock_on)
747     {
748         if(tmg_clockon() != SUCCESS)
749         {
750             lm_error("tmg_clear_error(): cannot turn clocks on\n");
751             return(FAILURE);
752         }
753         if(tmg_clockoff() != SUCCESS)
754         {
755             lm_error("tmg_clear_error(): cannot turn clocks off\n");
756             return(FAILURE);
757         }
758     }
759     return(SUCCESS);
760 }
761
762 /* the following functions were previously in tmg_cal.c */
763
764 tmg_play_til_key()
765 {
766     static char me[] = "tmg_play_til_key";
767     lm_message("PRESS ANY KEY TO CONTINUE...\n");
768     do {
769         if(tmg_play(TIMEOUT) != SUCCESS) {
770             tmg_report_failure(me, "tmg_play");
771             return(FAILURE);
772         }
773     } while (lm_check_key() == 0);
774     lm_message("THANK YOU\n");
775     while(lm_check_key() != 0) lm_get_key();
776     return SUCCESS;
777 }
778
779 long tmg_plot_delays()
780 {
781     INTR_INIT
782     register long ramp, edge;
783     long minramp = 0;
784     long minedge = 0;
785     long maxramp = NUMBER_OF_RAMPS - 1;
786     long maxedge = NUMBER_OF_EDGES + 1;
787     long decrement = 0;
788     diag_get_long(minramp, "start ramp", 0, maxramp);
789     diag_get_long(maxramp, "end ramp", minramp, maxramp);
790     diag_get_long(minedge, "start edge", 0, maxedge);
791     diag_get_long(maxedge, "end edge", minedge, maxedge);
792     diag_get_long(decrement, "decrement = 1, increment = 0", 0, 1);
793     INTR_BEGIN
794     tmg_set_test_mode(1); tmg_set_slot_count(1);
795     for (ramp = minramp; ramp <= maxramp; ++ramp) {
796         for (edge = minedge; edge <= maxedge; ++edge) {
797             tmg_plot_delay(ramp, edge, decrement);
798         }
799     }
800     INTR_END
801     tmg_restore_calibration();
802     intr();
803     INTR_END
804     tmg_restore_calibration();
805     return SUCCESS;
806 }
807
808 tmg_plot_delay(ramp, edge, decrement)
809 register u_long ramp, edge;
810 {
811     static char me[] = "tmg_plot_delay";
812     long p1, p2, c1, c2, junk;
813     register long thr, minth;
814     int nrtthr, stopthr, incthr;
815     tmg_set_slot_count(2);
816     switch (edge) {
817     default:
818         thr = tmg_predict_threshold(40000);
819         calib.EdgeSlope[ramp][edge], calib.EdgeOffset[ramp][edge];
820         minth = calib.EdgeMinThresh[ramp][edge];
821         break;
822     case 6: /* Edge 7 */
823         thr = (40000 - calib.SampleOffset[ramp][0] / calib.Edge7Slope[ramp];
824             - calib.SampleSlope[0] * 10) / calib.Edge7Slope[ramp];
825         minth = calib.Edge7MinThresh[ramp];
826         tmgptr->sample_mode = EDGE7SAMPLETRIGGERMODE;
827         break;
828     case 7: /* Sample Ramp */
829         thr = tmg_predict_threshold(40000);
830         calib.SampleSlope[ramp], calib.EarlySampleOffset[ramp];
831         minth = calib.SampleMinThresh[ramp];
832         tmgptr->sample_mode = EARLYSAMPLETRIGGERMODE;
833     }
834 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

diags/tmg\_util.c

DATE

5/23/89

PAGE #

TIME

4:41:37 pm

8/216

LINE # SOURCE TEXT

```

841     break;
842 }
843 if (thr < minth)
844     thr = minth;
845
846 if (decrement) {
847     startthr = 250;
848     stopthr = thr - 1;
849     incthr = -1;
850 } else {
851     startthr = thr;
852     stopthr = 251;
853     incthr = 1;
854 }
855
856 for (thr = startthr; thr != stopthr; thr += incthr) {
857     switch (edge) {
858     default:
859         if (! (p1 = tmg_find_period(ramp, edge, thr))) {
860             tmg_report_failure(me, "tmg_find_period");
861             return(FAILURE);
862         }
863         break;
864     case 6: /* Edge 7 */
865         if (! (p1 = tmg_find_sample_period(ramp, 0, thr, 10))) {
866             tmg_report_failure(me, "tmg_find_sample_period");
867             return(FAILURE);
868         }
869         break;
870     case 7: /* Sample Ramp */
871         if (! (p1 = tmg_find_sample_period(3, ramp, 255, thr))) {
872             tmg_report_failure(me, "tmg_find_sample_period");
873             return(FAILURE);
874         }
875         break;
876     }
877     if (tmg_get_next_lower_period(p1,
878         &p2, &junk, &junk, &junk, &junk) != SUCCESS) {
879         tmg_report_failure(me, "tmg_get_frequency_setting");
880         return FAILURE;
881     }
882     switch (edge) {
883     default:
884         o1 = tmg_predict_offset(p1, thr, calib.Edgeslope[ramp][edge]);
885         o2 = tmg_predict_offset(p2, thr, calib.Edgeslope[ramp][edge]);
886         break;
887     case 6: /* Edge 7 */
888         o1 = p1 - calib.Edge7Slope[ramp] * thr - calib.SampleSlope[0] * 10;
889         o2 = p2 - calib.Edge7Slope[ramp] * thr - calib.SampleSlope[0] * 10;
890         break;
891     case 7: /* Sample Ramp */
892         o1 = tmg_predict_offset(p1, thr, calib.SampleSlope[ramp]);
893         o2 = tmg_predict_offset(p2, thr, calib.SampleSlope[ramp]);
894         break;
895     }
896     lm_message("td td tld: t10d t10d t10d t10d\n", ramp, edge, thr,
897         p1, o1, p2, o2);
898 }
899
900 tmg_set_slot_count(1);
901 return SUCCESS;
902 }

```

Copyright 1989 Logic Modeling Systems		FILE bsp.diags/bsp.assem.s	DATE 5/23/89 TIME 4:41:07 pm	PAGE # 1/1
LINE #	TEXT			
1	SCCS_ID: bsp.assem.s rev 3.1, 4/24/89 at 07:54:58			
2	----- File Defines -----			
3				
4				
5				
6	=====			
7	DEFINITIONS SPECIFIC FOR THE CPU BOARD			
8	=====			
9				
10				
11	DISINT = 0x3780			
12	EMINT = 0x3800			
13	RTSCOPE = 0x81			
14	VRTX = 0x00			
15	IVT_BASE = 0x000000			
16	DUART_SR = 0x10c10014			
17	STATUS_REG_A = 0x10c10004			
18	STATUS_REG_B = 0x10c10014			
19	DUART_RX_TX_REG_A = 0x10c1000c			
20	DUART_RX_TX_REG_B = 0x10c1002c			
21	CHAR_TX_A = 24			
22	CHAR_RX_A = 25			
23	CHAR_TX_B = 28			
24	CHAR_RX_B = 29			
25	CPU_INTR_REG = 0x10c00001			
26	TIMER_CONTROL = 0x10c1000c			
27	TIMER3_COUNT = 0x10c10008			
28	READ_COUNTER3 = 0x00000000			
29	TIMER3_TOTAL = 0x013E			
30	TIMER3_LOW_COUNT = 0x00000000			
31	TIMER3_HIGH_COUNT = 0x01000000			
32	TIMER3_LOW_COUNT = 0x20000000			
33	TIMER3_HIGH_COUNT = 0x00000000			
34	ENABLE_LANCE = 0x00			
35	CPU_PMC_REG = 0x10c00001			
36	=====			
37				
38	VRTX and RTSCOPE function codes			
39	=====			
40				
41				
42	TR_ENTER = 0x0102			
43	TR_EXIT = 0x0103			
44	TR_ENTER = 0x0104			
45	TR_EXIT = 0x0105			
46	UI_TIMER = 0x0012			
47	UI_TIMER = 0x0013			
48	UI_TIMER = 0x0014			
49	UI_TIMER = 0x0015			
50	UI_TIMER = 0x0016			
51	UI_TIMER = 0x0017			
52	UI_TIMER = 0x0018			
53	UI_TIMER = 0x0019			
54	UI_TIMER = 0x001A			
55	UI_TIMER = 0x001B			
56	UI_TIMER = 0x001C			
57	UI_TIMER = 0x001D			
58	UI_TIMER = 0x001E			
59	UI_TIMER = 0x001F			
60	UI_TIMER = 0x0020			
61	UI_TIMER = 0x0021			
62	UI_TIMER = 0x0022			
63	UI_TIMER = 0x0023			
64	UI_TIMER = 0x0024			
65	UI_TIMER = 0x0025			
66	UI_TIMER = 0x0026			
67	UI_TIMER = 0x0027			
68	UI_TIMER = 0x0028			
69	UI_TIMER = 0x0029			
70	UI_TIMER = 0x002A			
71	UI_TIMER = 0x002B			
72	UI_TIMER = 0x002C			
73	UI_TIMER = 0x002D			
74	UI_TIMER = 0x002E			
75	UI_TIMER = 0x002F			
76	UI_TIMER = 0x0030			
77	UI_TIMER = 0x0031			
78	UI_TIMER = 0x0032			
79	UI_TIMER = 0x0033			
80	UI_TIMER = 0x0034			
81	UI_TIMER = 0x0035			
82	UI_TIMER = 0x0036			
83	UI_TIMER = 0x0037			
84	UI_TIMER = 0x0038			
85	UI_TIMER = 0x0039			
86	UI_TIMER = 0x003A			
87	UI_TIMER = 0x003B			
88	UI_TIMER = 0x003C			
89	UI_TIMER = 0x003D			
90	UI_TIMER = 0x003E			
91	UI_TIMER = 0x003F			
92	UI_TIMER = 0x0040			
93	UI_TIMER = 0x0041			
94	UI_TIMER = 0x0042			
95	UI_TIMER = 0x0043			
96	UI_TIMER = 0x0044			
97	UI_TIMER = 0x0045			
98	UI_TIMER = 0x0046			
99	UI_TIMER = 0x0047			
100	UI_TIMER = 0x0048			
101	UI_TIMER = 0x0049			
102	UI_TIMER = 0x004A			
103	UI_TIMER = 0x004B			
104	UI_TIMER = 0x004C			
105	UI_TIMER = 0x004D			
106	UI_TIMER = 0x004E			
107	UI_TIMER = 0x004F			
108	UI_TIMER = 0x0050			
109	UI_TIMER = 0x0051			
110	UI_TIMER = 0x0052			
111	UI_TIMER = 0x0053			
112	UI_TIMER = 0x0054			
113	UI_TIMER = 0x0055			
114	UI_TIMER = 0x0056			
115	UI_TIMER = 0x0057			
116	UI_TIMER = 0x0058			
117	UI_TIMER = 0x0059			
118	UI_TIMER = 0x005A			
119	UI_TIMER = 0x005B			
120	UI_TIMER = 0x005C			

Copyright 1989

Logic Modeling Systems

FILE

bsp.diags/bsp.assem.s

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

2/2

```

121      .globl _vtx_init
122      _vtx_init:
123          movl $VCTX_INIT,d0      Load Vctx Init Function code
124          trap $VCTX              Trap to Vctx
125          rts                     Return Error to C is DO
126
127      ----- Rtscope Init Interface
128
129      .globl _rts_init
130      _rts_init:
131      #ifdef RTSCOPE_DIAGS_DEBUG
132          link a6,0               frame pointer
133          movl a6,sp@-            save A0 on the stack
134          movl a6(0),a0           Rtscope const. table addr
135          movl $RTX_INIT,d0       DSINIT
136          trap $RTSCOPE           call Rtscope
137          movl sp@+,a0            restore A0
138          unlk a6                restore A6
139      #endif
140      rts
141
142      ----- Rtscope Go Interface
143
144      .globl _rts_go
145      _rts_go:
146      #ifdef RTSCOPE_DIAGS_DEBUG
147          movl $RTX_GO,d0         DRCO
148          trap $RTSCOPE           call Rtscope
149      #endif
150          movl $main,a0           New task address
151          movl $MAIN_TMODE,d1     Task mode
152          movb $MAIN_TID,d1       Task ID number
153          movb $MAIN_TPRI,d1      Task priority
154          movl $SC_CREATE,d0       System call code
155          trap $VCTX              Call VCTX
156
157      ----- Issue VCTX GO system call
158          movl $SC_GO,d0          SC_GO
159          trap $VCTX              Shouldn't come back
160          rts
161
162      _bad_start:
163          jmp _bad_start
164
165      =====
166      Interrupt Service Routines
167      =====
168
169      ----- Clock Board Error ISR
170
171      .globl _error_isr, _parity_isr
172      _error_isr:
173          movl d0,sp@-            dont save a7 & d0 save d1-d7,a0-a6
174          movl $0x7ffe,sp@-       This is our C interrupt routine
175          jar _mod_error_isr      restore above regs.
176          movl sp@+,0x7ffe
177
178      ----- Perform UI_EXIT from ISR through Vctx
179          movl $UI_EXIT,d0
180          trap $VCTX
181
182      ----- Parity ISR
183
184      _parity_isr:
185          movl d0,sp@-            dont save a7 & d0 save d1-d7,a0-a6
186          movl $0x7ffe,sp@-       This is our C interrupt routine
187          jar _parity_error       restore above regs.
188          movl sp@+,0x7ffe
189
190      ----- Perform UI_EXIT from ISR through Vctx
191          movl $UI_EXIT,d0
192          trap $VCTX
193
194      ----- Ethernet ISR
195
196      .globl _ether_isr
197      _ether_isr:
198          movl d0,sp@-            dont save a7 & d0 save d1-d7,a0-a6
199          movl $0,system_call     check if a system call is made?
200          jar _lance_isr          This is our C interrupt routine
201          movl _system_call, d0    let us check bit 0
202          movl sp@+,0x7ffe        restore above regs.
203          btst $0,d0             should we exit thru VCTX
204          beq no_sys_calls        go to no system calls, if no VCTX
205                                  calls were made.
206
207      ----- Perform UI_EXIT from ISR through Vctx
208          movl $UI_EXIT,d0
209          trap $VCTX
210
211      ----- NO SYSTEM CALLS WERE MADE -----
212
213      no_sys_calls:
214          movl sp@+,d0
215          rts
216
217      ----- DUART ISR
218      ----- Chasael A Serial Port - RX buffer full -----
219
220      .globl _serial_isr
221      _serial_isr:
222          movl d0,sp@-            dont save a7 & d0 save d1-d7,a0-a6
223          movl d1,sp@-            check if a system call is made?
224          movl a0,sp@-            This is our C interrupt routine
225          movl $DUART_SR,a0       let us check bit 0
226          | get status

```

Copyright 1989 Logic Modeling Systems		FILE bsp.diags/bsp.assem.s	DATE 5/23/89 TIME 4:41:07 pm	PAGE # 3/3
LINE #	TEXT			
241	movl a0,d1	load into reg		
242	andl _uart_mask,d1	only look @ interrupts we are interested in		
243	andl 0x00000000,d1	see if VRTX interrupt		
244	jeq rtscops_serial_isr	no? go to RTscops		
245	movl a0,d1	load into reg		
246	bstat %CHAR_RX_A,d1	check for CHAR_RX_A		
247	beq not_tx_a	nothing rxd on channel a		
248	movl \$STATUS_REG_A,a0	get the status		
249	movl a0,d0	status is in 24-31		
250	movl \$UART_RX_TX_REG_A,a0	get the byte		
251	movl a0,d1	byte is in 24-31		
252	andl 0x00000000,d0	see if error to rxd, break or framing		
253	beq err_rx_a	process error		
254	movq \$24,d0	shift 24 bits		
255	movq d0,d1	d1 bits 0-7 = data		
256				
257	cmpl \$XOFF,d1	Flow control check for XON & XOFF		
258	jne not_xoff			
259	movb 1,_XOFF			
260	jar txa_exit			
261	bna not_tx_a			
262	not_xoff:			
263	cmpl \$XON,d1			
264	jne not_xon			
265	movb 0,_XOFF			
266	bna not_tx_a			
267	not_xon:			
268				
269	movl \$UI_RXCH,d0			
270	trap \$VRTX			
271	not_rx_a:			
272	movl \$UART_SR,a0	get status		
273	movl a0,d1	load into reg		
274	bstat %CHAR_TX_A,d1	check for CHAR_TX_A		
275	beq not_tx_a	nothing to tx on channel a		
276	jar txa_isr	go and transmit		
277	not_tx_a:			
278	movl ap0,a0	restore address register		
279	movl ap0,d1			
280	movl \$UI_EXIT,d0			
281	trap \$VRTX			
282	err_rx_a:			
283	andl 0x00000000,d0	see if break		
284	beq not_break	not break		
285	movq \$24,d0	shift 24 bits		
286	movq d0,d1	d1 bits 0-7 = data		
287	beq not_tx_a	not break start		
288	jar txa_isr_bna	process break		
289	bna not_tx_a	continue		
290				
291	END txa_isr			
292				
293				
294	Channel A Serial Port - TX buffer empty			
295				
296	global txa_isr, Vrtx_txa			
297	txa_isr:			
298	cmpl \$0,d1	Check flow control is not on		
299	jne no_flow_ctrl			
300	bna txa_exit			
301	no_flow_ctrl:			
302				
303	movl \$UI_TXCH,d0			
304	trap \$VRTX			
305	cmpl \$0,d0			
306	bna txa_exit			
307				
308				
309	----- This is the VRTX TX routine -----			
310	----- d1 = data to TX -----			
311	----- save A0 & D0 -----			
312	----- This is the hook for VRTX to resume or start transmitting chars -----			
313	Vrtx_txa:			
314	movl d0,ap0	get address of register		
315	movl a0,ap0	shift 24 bits to get byte in upper bits		
316	movl \$UART_RX_TX_REG_A,a0	d1 bits 00-07 = data		
317	movq \$24,d0	byte is in 24-31 after shift		
318	andl d0,d1	get status reg contents		
319	movl d1,a0	this is a write only reg. so read from memory		
320	movl \$UART_SR,a0	d1 = 03 enable tx ints		
321	movl _uart_mask,d1	save in memory		
322	orl 0x00000000,d1	load into reg		
323	movl d1,_uart_mask	restore registers		
324	movl d1,a0			
325	movl ap0,a0			
326	movl ap0,d0			
327	movl ap0,d0			
328	txa_exit:			
329	movl \$UART_SR,a0	get status reg contents		
330	movl _uart_mask,d1	this is a write only reg. so read from memory		
331	andl 0x00000000,d1	d1 = disable tx ints		
332	movl d1,_uart_mask	save in memory		
333	movl d1,a0	load into reg		
334	movl d1,a0			
335	movl d1,a0			
336	END txa_isr			
337				
338	RTscops serial int. on chas B			
339				
340	rtscops_serial_isr:			
341	movl a0,d1	load into reg		
342	bstat %CHAR_RX_B,d1	check for CHAR_RX_B		
343	beq not_tx_b	nothing rxd on channel b		
344	jar _modem_rx	receive character		
345	not_tx_b:			
346	movl \$UART_SR,a0	get status		
347	movl a0,d1	load into reg		
348	bstat %CHAR_TX_B,d1	check for CHAR_TX_B		
349	beq not_tx_b	nothing can be transmitted on channel b		
350	jar _modem_tx	transmit character		
351	not_tx_b:			
352	movl ap0,a0	restore address register		
353	movl ap0,d1			
354	movl \$UI_EXIT,d0			
355	trap \$VRTX			
356	END rx_b_isr			
357				
358				
359	----- This is the RTSCOPS TX routine -----			
360	----- d1 = data to TX -----			

Copyright 1989 Logic Modeling Systems		FILE bsp.diags/bsp.assem.s	DATE 5/23/89 TIME 4:41:07 pm	PAGE # 4/4
--	--	-------------------------------	---------------------------------------	---------------

LINE #	TEXT
361	----- save A0 & D0 -----
362	----- This is the hook for RTSCOPE to resume or start transmitting chars --
363	
364	.globl _Vrtx_txh
365	_Vrtx_txh:
366	rts
367	----- end txh_isr -----
368	
369	.globl _rtscope_in_poll, _rtscope_out_poll
370	_rtscope_in_poll:
371	movl    \$a0,\$a0
372	movl    \$d1,\$d1
373	movl    \$0xfffffff,\$d0

Copyright 1989 Logic Modeling Systems		FILE bsp.diags/bsp.assem.s	DATE 5/23/89	PAGE # 5/5
			TIME 4:41:07 pm	
LINE #	TEXT			
481	movl a0,ap0-			
482	movl a0,ap0-			
483	movl a0,ap0-			
484	movl a0,ap0-			
485				
486				
487				
488	-- disable interrupts			
489				
490	movl SCPS_INTX_REG,a0			
491	movb a0,a0			
492	andb 0x7f,a0			
493				
494				
495	-- reload tick value			
496				
497	movl STIMER_CONTROL,a0			
498	movl STIMER_COUNTER,a0			
499	movl STIMER2_COUNT,a0			
500	movl a0,d1			
501	movl 0x24,d0			
502	sarl d0,d1			
503	andl 0x7f,d1			
504	movl a0,d0			
505	sarl d0,d0			
506	sarl d0,d0			
507	andl 0x7f,d0			
508	andl 0x7f,d0			
509	andl 0x7f,d0			
510	addw STIMER2_TOTAL,d0			
511	slll d0,d0			
512	slll d0,d0			
513	movl d0,d1			
514	slll d0,d0			
515	movl a0,ap0			
516	movl d1,ap0			
517	movl STIMER2_LOW_COUNT,a0			
518				
519	movl STIMER2_HIGH_COUNT,a0			
520	of ticks since beginning of time			
521				
522	-- Enable interrupts			
523				
524	movl SCPS_INTX_REG,a0			
525	btst 07,d1			
526	breq disabled_ints			
527	orl 0x00,a0			
528	disabled_ints:			
529	addl \$1,im_tick			
530	addw \$1,housekeeping			
531	cmpr \$00,housekeeping			
532	jne no_housekeeping			
533	clrw housekeeping			
534	movl \$SC_IPOST,d0			
535	movl timer_semaphore,d1			
536	trap \$VRTX			
537	no_housekeeping:			
538				
539	----- Make VI_TIMER Call to Vrtx			
540				
541				
542	- inform VRTX of tick			
543	movl \$VI_TIMER,d0			
544	trap \$VRTX			
545				
546	movl ap0+,d1			
547	movl ap0+,d1			
548	movl ap0+,a0			
549	restore registers			
550	restore registers			
551				
552	----- Perform VI_EXIT from ISR through Vrtx			
553	movl \$VI_EXIT,d0			
554	trap \$VRTX			
555				
556	END ct_isr			
557				
558				
559	----- PROBE BUS ERROR HANDLER			
560				
561				
562	.globl _bus_isr			
563	_bus_isr:			
564	movl sp,_bus_error_address			
565				
566	moveml 0x7ffe,ap0-			
567	lax _bus_error			
568	moveml ap0+,0x7fff			
569	rts			
570	dont save a7 save d1-d7,a0-a6			
571	restore above regs.			
572				
573	----- DIAG BUS ERROR HANDLER			
574				
575				
576	.globl _diag_bus_isr			
577	_diag_bus_isr:			
578	movl sp,_bus_error_address			
579				
580	moveml 0x7ffe,ap0-			
581	lax _diag_bus_error			
582	moveml ap0+,0x7fff			
583	rts			
584	dont save a7 save d1-d7,a0-a6			
585	restore above regs.			
586				
587				
588	----- TRAP ROUTINES for RTscope and VRTX			
589				
590				
591				
592	.globl _trp_vrtx			
593	_trp_vrtx:			
594	trap \$VRTX			
595	rts			
596				
597				
598	.globl _trp_rtscope			
599	_trp_rtscope:			
600				
601	trap \$RTSCOPE			
602	rts			
603				
604				
605				
606				



Copyright 1989  
Logic Modeling Systems

FILE  
bsp.diags/bsp.assem.s

DATE 5/23/89  
TIME 4:41:07 pm

PAGE #  
6/6

```

LINE #      TEXT
601      ----- RESET_BSD
602      cause a board reset.
603
604      ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
605
606      .globl _reset_brd
607      _reset_brd:
608      .reset
609      .jar    bsp_start
610
611      board reset occurs here
612
613      rts
614
615
616      .globl _do_nothing
617      _do_nothing:
618      rts
619
620
621      ----- Create a Supervisory task -----
622      err = sc_tcreate_supv_set_boot( task_address, SUPV\USER task , task_id, task priority)
623      if( err != 0 ) FAILURE;
624
625      .globl _sc_tcreate_supv_set_boot
626      _sc_tcreate_supv_set_boot:
627      link    a6, #0
628      movl    a0, sp0
629      movl    d1, sp0
630      movl    d1, sp0
631      movl    d1, sp0
632      movl    d1, sp0
633      movl    #0, d1
634      movl    #0, d1
635      movl    a6@($), a0      task address
636      movl    a6@($xc), d3    task mode
637      movl    a6@($x10), d2   task id
638      movl    a6@($x14), d1    task priority
639      movl    $SC_TCREATE, d0  System call code
640      trap    $VRTX          Call VRTX
641      movl    sp0+, d3
642      movl    sp0+, d2
643      movl    sp0+, d1
644      movl    sp0+, a0
645      halt    a6
646      rts
647
648      .globl _disable_cache
649      _disable_cache:
650      movl    #8, d0
651      movc    d0, cacr
652      rts

```

Clear and disable instruction cache

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

1/7

```

1  // SCCS ID: bsp.c rev 3.1, 4/24/89 at 07:55:02
2  //
3  // This is part of the board support package
4  //
5  // Make sure that both bsp.lm500/bsp.c and bsp.diags/bsp.c
6  // are exactly the same.
7  //
8  //
9  #include "common.h"
10 #include "task.h"
11 #include "uart.h"
12 #include "lm_rst_wr.h"
13 #include "lm500.h"
14 #include "cpu.h"
15 #include "cpu_vec.h"
16 #include "uart_err.h"
17 #include "swram.h"
18 #include "id.h"
19 //
20 // External routine definitions
21 //
22 extern int init_sys();
23 extern void serial_isr();
24 extern void parity_isr();
25 extern void ct_isr();
26 extern void reset_cpu();
27 extern void other_isr();
28 extern void error_isr();
29 extern void vtx_cha();
30 extern void vtx_tmb();
31 extern void rtscop_in_poll();
32 extern void rtscop_out_poll();
33 extern void void_id_load();
34 extern void trp_vtx();
35 extern void trp_rtscop();
36 extern void u_short_lm_swram_access();
37 extern void u_long_network_timeout();
38 extern void reset_hw();
39 //
40 #define MICROSECS_PER_SECOND 1000
41 //----- The Globals -----
42 //-----
43 //-----
44 //-----
45 //-----
46 //-----
47 //-----
48 //-----
49 //-----
50 //-----
51 //-----
52 //-----
53 //-----
54 //-----
55 //-----
56 //-----
57 //-----
58 //-----
59 //-----
60 //-----
61 //-----
62 //-----
63 //-----
64 //-----
65 //-----
66 //-----
67 //-----
68 //-----
69 //-----
70 //-----
71 //-----
72 //-----
73 //-----
74 //-----
75 //-----
76 //-----
77 //-----
78 //-----
79 //-----
80 //-----
81 //-----
82 //-----
83 //-----
84 //-----
85 //-----
86 //-----
87 //-----
88 //-----
89 //-----
90 //-----
91 //-----
92 //-----
93 //-----
94 //-----
95 //-----
96 //-----
97 //-----
98 //-----
99 //-----
100 //-----
101 //-----
102 //-----
103 //-----
104 //-----
105 //-----
106 //-----
107 //-----
108 //-----
109 //-----
110 //-----
111 //-----
112 //-----
113 //-----
114 //-----
115 //-----
116 //-----
117 //-----
118 //-----
119 //-----
120 //-----

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
bsp.diags/bsp.c

DATE 5/23/89  
TIME 4:41:07 pm

PAGE #  
2/8

```

LINE #          SOURCE TEXT
121          /*
122          (void)lm_vxram_access(lmodeler_state, MODELER_STATE, SIZEOF_MODELER_STATE, MEMORY_WRITE, (u_long *) &err);
123          Uarta_inx();
124          Uartb_inx();
125          /*
126          set network timeout
127          /*
128          if(lm_vxram_access(&short_network_timeout, NETWORK_TIMEOUT, SIZEOF_NETWORK_TIMEOUT, MEMORY_READ, (u_long *) &err) == FAILURE)
129          {
130              sys_out("Unable to read network timeout\n");
131          }
132          /*
133          Turn seconds to milliseconds
134          /*
135          network_timeout = short_network_timeout;
136          network_timeout *= NSECONDS_PER_SECOND;
137          /*
138          read id from
139          /*
140          id_load((u_char *)CPU_ID_PROM, (u_char *)(&id_prom)); /* Fetch id prom */
141          enable_cpu_regs(); /* enable CPU regs */
142          rts_go(); /* Execute RTscope GO command */
143          /*
144          /*
145          /* SHOULD NOT COME BACK: if it does ??????????
146          /* do nothing.
147          /*
148          for(;;)
149              reset_cpu(SUICIDE);
150          /*
151          /* end init_sys */
152          /*
153          /*
154          /*
155          /*
156          /* Create VRTX configuration table
157          /*
158          /*
159          init_vrtxtbl()
160          {
161          /*
162          VRTX_CMTBL *ct = (VRTX_CMTBL *)(&(cftbl->v_cft));
163          /*
164          ct->v_wspsc = VRTX_WK_SPACE; /* VRTX workspace address */
165          ct->v_wspsc_size = VRTX_WK_SIZE; /* VRTX workspace size */
166          ct->sys_stk_size = SYS_STK; /* System stack size */
167          ct->lar_stk_size = LAR_STK; /* Interrupt stack size */
168          ct->cb_count = CB_COUNT; /* Number of VRTX CB */
169          ct->real = 0; /* Reserved */
170          ct->real2 = 0; /* Reserved */
171          ct->dis_level = VC_DIS_LEVEL; /* Component disable level */
172          ct->usr_stk_size = USR_STK; /* User stack size */
173          ct->real3 = 0; /* Reserved */
174          ct->usr_tsk_count = TASK_COUNT; /* Number of tasks */
175          ct->real4 = 0; /* Reserved */
176          ct->tx_rdy = (long) Vrtx_tx; /* Transmit ready interrupt */
177          ct->txcreate = 0; /* User supplied txcreate */
178          ct->txdelete = 0; /* User supplied txdelete */
179          ct->txwatch = 0; /* User supplied txwatch */
180          #ifdef RTSCOPE_DIAGS_DEBUG
181          ct->cvt_add = (CVT_TBL *)(&(cftbl->cvt)); /* Component conf. table add */
182          #else
183          ct->cvt_add = (CVT_TBL *) 0; /* Component conf. table add */
184          #endif
185          /*
186          /* end init_vrtxtbl */
187          /*
188          /*
189          /*
190          /*
191          /* Create RTscope configuration table
192          /*
193          /*
194          init_rtsftbl()
195          {
196          /*
197          RTCT_CMTBL *ct = (RTCT_CMTBL *)(&(cftbl->r_cft));
198          RTIO_CMTBL *ioct = (RTIO_CMTBL *)(&(cftbl->rio_cft));
199          short
200          {
201          /* RTscope configuration table */
202          ct->db_vrtx_c = VRTX_BASE; /* VRTX entry point */
203          ct->db_rt_w = RTX_WK_SPACE; /* RTscope workspace addr */
204          ct->db_ws_size = RTX_WK_SIZE; /* RTscope workspace size */
205          ct->db_illegal = 0x0000; /* Illegal instruction, 4802 */
206          ct->db_trap_t = 0; /* User supplied routine */
207          ct->db_comp_dis = VC_DIS_LEVEL; /* Component disable level */
208          ct->db_priority = RT_SCP_PRI; /* RTscope task priority */
209          ct->db_tid = RT_SCP_TID; /* RTscope task ID number */
210          ct->db_trap = (long)trp_vrtx; /* User supplied routine TRAP */
211          ct->db_io_conf = ioct; /* IO conf. table address */
212          for( i = 0; i < 4; i++) /* Reserved */
213              ct->real[i] = 0;
214          ct->db_vrtx_d = VRTX_WK_SPACE; /* VRTX workspace address */
215          ct->db_trp_vrtx = TRP_VRTX; /* VRTX TRAP vector number */
216          for( i = 0; i < 3; i++) /* Reserved */
217              ct->real2[i] = 0;
218          /*
219          /* RTscope IO configuration table */
220          ioct->db_gid = 1; /* Queue ID number */
221          ioct->db_qsize = 80; /* Queue size */
222          ioct->db_inpt_ll = 80; /* Input line length */
223          ioct->db_hstack = 10; /* History lines */
224          ioct->db_alias = 20; /* Number of alias */
225          ioct->db_sys_tab = 20; /* Number of symbols */
226          ioct->db_sum_chal = 2; /* Number of ports */
227          ioct->db_toggle = 0x01; /* Toggle character */
228          ioct->db_xoff = 0x01; /* Xoff */
229          ioct->db_xoff2 = 0x01; /* Xoff */
230          ioct->db_exit = 0x01; /* Host exit character */
231          ioct->db_txdy = (long)Vrtx_tx; /* Hyperlink TXDY driver addr */
232          ioct->db_in_filter = 0; /* Input filter */
233          ioct->db_out_filter = 0; /* Output filter */
234          ioct->db_pputcl = (long)rtscope_out_poll; /* Hyperlink input poll */
235          ioct->db_pgetcl = (long)rtscope_in_poll; /* Hyperlink output poll */
236          ioct->db_pputcl2 = 0; /* Host input poll */
237          ioct->db_pgetcl2 = 0; /* Host output poll */
238          ioct->db_spec_keys = (DB_SPEC *) 0; /* Special keys table */
239          /*
240          /* end init_rtsftbl */

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

3/9

```

LINE #          SOURCE TEXT
241
242
243 /*-----*/
244 /*          init_cvt          */
245 /* Create component configuration table */
246 /*-----*/
247
248
249 init_cvt()
250 {
251     CVT_TBL      *ct = (CVT_TBL *)(&(cftbl->cvt));
252     short
253
254     ct->hr_max = HR_MAX,          /* Ready System highest component no. */
255     ct->usr_max = USR_MAX,        /* User highest component no. */
256
257     for( i = 0, i < 6; i++ )      /* Reserved */
258         ct->res1[ i ] = 0,
259     for( i = 0, i < 8; i++ )      /* Reserved */
260         ct->res2[ i ] = 0,
261
262     ct->compl_code = 0,           /* No IOX */
263     ct->compl_work = 0,           /* No IOX */
264     ct->compl_code = 0,           /* No IPI */
265     ct->compl_work = 0,           /* No IPI */
266
267 } /* end init_cvt */
268
269
270
271
272 /*-----*/
273 /*          setup_ivt          */
274 /* Setup interrupt vector table */
275 /*-----*/
276
277 setup_ivt()
278 {
279     unsigned long *ivt = (unsigned long *)VEC_BASE_ADDRESS,
280
281     /*
282     * The bus handler
283     */
284     *(ivt + VEC_BUS_ERROR) = (unsigned long) bus_isr,
285     *(ivt + VEC_ADDRESS_ERROR) = (unsigned long) bus_isr,
286     *(ivt + VEC_ILLEGAL_INSTRUCTION) = (unsigned long) bus_isr,
287
288     /*
289     * Set up rtoscope entry
290     */
291     *(ivt + 0x11) = (RTS_BASE + RTS_ENTRY), /* RTscope entry point */
292
293     /*
294     * Set up Vtr vectors
295     */
296     *(ivt + 0x40) = (long)&(cftbl->v_cftbl), /* Config Table */
297     *(ivt + 0x20) = VRTX_BASE, /* VRTX Base */
298
299     /*
300     * Set up serial port interrupt vectors
301     */
302     /* Tx/Rx ISR handler address to IVT entry vector level 3 */
303     *(ivt + VEC_DUART_INTERRUPT) = (long)serial_isr,
304
305     /*
306     * Set up Counter/Timer interrupt vectors
307     */
308     /* Count/Timer ISR address to IVT 35, Auto vector level 4 */
309     *(ivt + VEC_TIMER_INTERRUPT) = (unsigned long)ct_isr,
310
311     /*
312     * Set up ethernet interrupt vector
313     */
314     /* ethernet ISR address to IVT 10, Auto vector level 6 */
315     *(ivt + VEC_LANCE_DOORBELL_INTE) = (unsigned long)ether_isr,
316
317     /*
318     * Clock board error ISR
319     */
320     *(ivt + VEC_CB_INTERRUPT) = (unsigned long)error_isr,
321
322     /*
323     * Parity ISR, IVT 31, MMIO
324     */
325     *(ivt + VEC_PARITY_INTERRUPT) = (unsigned long)parity_isr,
326
327 } /* end setup_ivt */
328
329
330
331
332 /*
333 * Enable cpu control registers
334 */
335 enable_cpu_reg()
336 {
337     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
338
339     /*
340     * enable timer gates, enable the timer interrupt
341     */
342     p_cpu_ctl_reg->timer_gate_0 = 1,
343     p_cpu_ctl_reg->timer_gate_1 = 1,
344     p_cpu_ctl_reg->timer_intr_ena_2 = 1,
345
346     /*
347     * enable parity intr
348     */
349     p_cpu_ctl_reg->parity_intr_ena = 1,
350
351     /*
352     * enable interrupts
353     */
354     p_cpu_ctl_reg->global_intr_ena = 1,
355
356 }
357
358 /*
359 * reset the CPU
360 */
361 void
362 reset_cpu(reset_modeler_state)
363 char reset_modeler_state;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/bsp.c

DATE

5/23/89

PAGE #

TIME

4:41:07 pm

4/10

LINE #

SOURCE TEXT

```
361 {
362     char modeler_reset = reset_modeler_state;
363     u_long err;
364     cpu_control_reg_struct *cpu_ctl = (cpu_control_reg_struct *) CPU_CONTROL_REG;
365
366     /* release for reset */
367     /*
368     */
369     if( reset_modeler_state != REBOOT )
370     {
371         (void)lm_wvarm_access(&modeler_reset, MODELER_RESET, sizeof(MODELER_RESET), MEMORY_WRITE, (u_long *) &err);
372         reset_modeler_state = SHUTDOWN;
373     }
374     (void)lm_wvarm_access(&reset_modeler_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err);
375     cpu_ctl->suicide = 1;
376     for(;;)
377 }
```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/modem.c

DATE

5/23/89

PAGE #

TIME

4:41:08 pm

1/11

SOURCE TEXT

```

1  /* SOCS_ID: modem.c rev 3.1, 4/24/89 at 07:55:06 */
2  #include "common.h"
3  #include "cpu_ver.h"
4  #include "m_rd_wr.h"
5  #include "mod_rsr.h"
6  #include "svaRam.h"
7
8  #define Duart_ar      (*((int *)0x10c20014))
9  #define Srb           (*((int *)0x10c20024))
10 #define Uart_b        (*((int *)0x10c2002c))
11
12 #define GLOCK(q)      ((q)->qlock)
13 #define PUTPARK(q)    ((q)->putpark)
14 #define GETPARK(q)    ((q)->getpark)
15 #define ENPTY(q)      ((q)->backp == (q)->frontp)
16
17 #define UART_BUF_SIZE 256
18
19 struct qcontrol {
20     char *qlock,
21     int backp,
22     int frontp,
23     char *getpark,
24     int getwaiters,
25     char *putpark,
26     int putwaiters,
27     char qbuf[UART_BUF_SIZE],
28 };
29
30 struct qcontrol Modem_txq, Modem_rxq;
31
32 static int *Save_duart_lsr;
33
34 #define read_duart_status() *((int *)0x10c20004)
35
36 modem_init()
37 {
38     qinit(&Modem_txq);
39     qinit(&Modem_rxq);
40 }
41
42 modemprintline(s)
43 char *s;
44 {
45     while (*s) {
46         qputc(&Modem_txq, *s++);
47     }
48     flushmodem();
49 }
50
51 getmodem()
52 {
53     extern int Uart_mask;
54
55     /* this txq stuff should be in housekeeping task */
56     if (!ENPTY(&Modem_txq)) {
57         Uart_mask |= 0x10000000;
58         Duart_ar = Uart_mask;
59     }
60
61     return qgetc(&Modem_rxq);
62 }
63
64 getmodem_tmo(tmo, err)
65 int tmo, *err;
66 {
67     extern int Uart_mask;
68
69     /* this txq stuff should be in housekeeping task */
70     if (!ENPTY(&Modem_txq)) {
71         Uart_mask |= 0x10000000;
72         Duart_ar = Uart_mask;
73     }
74
75     return qgetc_tmo(&Modem_rxq, tmo, err);
76 }
77
78 flushmodem()
79 {
80     if (!ENPTY(&Modem_txq)) {
81         Uart_mask |= 0x10000000;
82         Duart_ar = Uart_mask;
83     }
84 }
85
86 putmodem(c)
87 int c;
88 {
89     qputc(&Modem_txq, c);
90     Uart_mask |= 0x10000000;
91     Duart_ar = Uart_mask;
92 }
93
94 modem_rx()
95 {
96     register int c;
97     register int srb;
98     c = srb;
99
100     register struct qcontrol *qp = &Modem_rxq;
101
102     srb = Srb;
103     c = Uart_b >> 24;
104     if (srb & 0x00000000) {
105         /* is this a break? */
106         if (srb & 0x80000000) {
107             Uartb_rx_brk();
108         }
109         return;
110     }
111     if (qfull(qp)) return; /* waste character */
112     qp->qbuf[qp->backp++] = c;
113     if (qp->backp == UART_BUF_SIZE) qp->backp = 0;
114     if (qp->getwaiters != 0) {
115         qp->getwaiters--;
116         ac_post(GETPARK(qp), (char *)1, &err);
117     }
118 }
119
120

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

bsp.diags/modem.c

DATE

5/23/89

PAGE #

TIME

4:41:08 pm

2/12

## SOURCE TEXT

```

121 modem_tx()
122 {
123     int err;
124     extern int uart_mask;
125     register struct qcontrol *qp = &modem_c;
126
127     if (!EMPTY(qp)) {
128         uart_b = qp->dbuf(qp->frontp++) << 24;
129         if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
130         if (qp->putwaiters != 0) {
131             qp->putwaiters--;
132             ac_post(PUTPARK(qp), (char *)1, &err);
133         }
134         uart_mask |= 0x10000000;
135         uart_ar = uart_mask;
136     } else {
137         uart_mask &= 0x0f000000;
138         uart_ar = uart_mask;
139     }
140 }
141
142
143 qinit(qp)
144 struct qcontrol *qp;
145 {
146     qp->qlock = (char *)1;
147     qp->backp = 0;
148     qp->frontp = 0;
149     qp->getpark = 0;
150     qp->getwaiters = 0;
151     qp->putpark = 0;
152     qp->putwaiters = 0;
153 }
154
155 qputc(qp, c)
156 struct qcontrol *qp;
157 {
158     int err;
159
160     ac_post(QLOCK(qp), 0, &err);
161     while (qfull(qp)) {
162         ++qp->putwaiters;
163         ac_post(QLOCK(qp), (char *)1, &err);
164         ac_post(PUTPARK(qp), 0, &err);
165         ac_post(QLOCK(qp), 0, &err);
166     }
167     qp->dbuf(qp->backp++) = c;
168     if (qp->backp >= UART_BUFSIZE) qp->backp = 0;
169     if (qp->getwaiters != 0) {
170         qp->getwaiters--;
171         ac_post(GETPARK(qp), (char *)1, &err);
172     }
173     ac_post(QLOCK(qp), (char *)1, &err);
174     if (c == (int)'\n')
175         qputc(qp, (int)'\r'); /* Recursive Cool */
176 }
177
178 qgetc(qp)
179 struct qcontrol *qp;
180 {
181     int c, err;
182
183     ac_post(QLOCK(qp), 0, &err);
184     while (EMPTY(qp)) {
185         ++qp->getwaiters;
186         ac_post(QLOCK(qp), (char *)1, &err);
187         ac_post(GETPARK(qp), 0, &err);
188         ac_post(QLOCK(qp), 0, &err);
189     }
190     c = qp->dbuf(qp->frontp++);
191     if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
192     if (qp->putwaiters != 0) {
193         qp->putwaiters--;
194         ac_post(PUTPARK(qp), (char *)1, &err);
195     }
196     ac_post(QLOCK(qp), (char *)1, &err);
197     return c;
198 }
199
200 /* get c with timeout */
201 qgetc_two(qp, two, err)
202 struct qcontrol *qp;
203 int two, *err;
204 {
205     int c;
206
207     ac_post(QLOCK(qp), two, err);
208     if (*err == 0) return 0;
209     while (EMPTY(qp)) {
210         ++qp->getwaiters;
211         ac_post(QLOCK(qp), (char *)1, err);
212         ac_post(GETPARK(qp), two, err);
213         if (*err == 0) return 0;
214         ac_post(QLOCK(qp), two, err);
215         if (*err == 0) return 0;
216     }
217     c = qp->dbuf(qp->frontp++);
218     if (qp->frontp >= UART_BUFSIZE) qp->frontp = 0;
219     if (qp->putwaiters != 0) {
220         qp->putwaiters--;
221         ac_post(PUTPARK(qp), (char *)1, err);
222     }
223     ac_post(QLOCK(qp), (char *)1, err);
224     return c;
225 }
226
227 qfull(qp)
228 struct qcontrol *qp;
229 {
230     int n;
231     n = qp->frontp - qp->backp - 1;
232     return ((n == 0) || (n == -UART_BUFSIZE));
233 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

tasks.diahs/hkeeping\_task.c

DATE

5/23/89

PAGE #

TIME

4:42:31 pm

1/1

```

1  // SCCS ID: hkeeping_task.c rev 3.1, 4/24/89, at 07:55:11
2
3  #include "common.h"
4  #include "cpu.h"
5  #include "network.h"
6  #include "lanes.h"
7  #include "md_err.h"
8  #include "nvars.h"
9
10 unsigned long timer_semaphore;
11 extern u_short check_connection_for_life();
12 extern u_long ln_tick;
13 extern CONNECTION *table_of_conns();
14 extern u_long network_timeout;
15 extern BOOT_STRUCT boot;
16 static char string[ 50 ];
17 #define MAX_TRIES 3
18
19 void
20 housekeeping_task()
21 {
22     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
23     int err;
24     u_char value=0;
25     #ifdef BROKEN_HARDWARE
26     extern char reinitialize_lanec;
27     #endif
28     #ifdef BROKEN_HARDWARE
29     u_char users = 0;
30     register CONNECTION *conn_table;
31     register CONNECTION *conn;
32     /*
33     * This loop is the house keeping task.
34     */
35     while(1)
36     {
37         ac_spin( timer_semaphore, 0, &err);
38         if(err) printf("error in spin in housekeeping tx.", err);
39         if( value == 1)
40         {
41             p_cpu_ctl_reg->net_test_led = net_LED_OFF;
42             value = 0;
43         }
44         else
45         {
46             p_cpu_ctl_reg->net_test_led = net_LED_ON;
47             value = 1;
48         }
49     }
50     #ifdef BROKEN_HARDWARE
51     if( reinitialize_lanec == 1)
52     {
53         reinitialize_lanec = 0;
54         (void)ac_lock();
55         if( get_lanec_ready_to_go() != SUCCESS)
56             printf("Can't reinitialize lanec\n");
57         else
58             printf("Reinitialized lanec\n");
59         if( start_lanec() != SUCCESS)
60             printf("Can't start lanec\n");
61         else
62             printf("Restarted lanec\n");
63         (void)ac_unlock();
64     }
65     #endif
66     /* go thru conn structures
67     * checking for timeouts
68     */
69     conn_table = table_of_conns( 0 );
70     for( users = 0; users < MAX_USERS; users++ )
71     {
72         conn = *conn_table++;
73         CPU_DISABLE_INTERRUPTS;
74         /*
75         * close the connection
76         */
77         if( conn->do_close == TRUE )
78         {
79             if( close_connection_for_server( conn ) == FAILURE ) {
80                 sprintf(string, "Unable to close connection %d\n", conn->fd);
81                 output_routine(string);
82             }
83             else
84             {
85                 /* did we send the last part of the reply
86                 * and is this connection supposed to close
87                 */
88                 if( conn->am_sending == FALSE && conn->am_closing == TRUE ) {
89                     conn->do_close = TRUE;
90                 }
91             }
92         }
93         if( network_timeout != 0 && conn != (CONNECTION *) NULL && conn->am_timing_out == TRUE )
94         {
95             if( conn->time_to_live < (ln_tick * 5) )
96             {
97                 if( check_connection_for_life( conn ) == FAILURE )
98                 {
99                     sprintf(string, "Can't check connection for life in housekeeping task user %d\n", users);
100                     output_routine(string);
101                 }
102                 conn->time_to_live = (ln_tick * 5) + network_timeout;
103                 if( conn->number_of_live_retries > MAX_TRIES )
104                 {
105                     /*
106                     * The host daemon is not responding to daemon
107                     * requests. This does not mean that the host is dead,
108                     * so just make a note of it and continue.
109                     */
110                     sprintf(string, "Warning: Connection timeout for user: %d\n", users);
111                     output_routine(string);
112                     sprintf(string, "Check host daemon.\n");
113                     output_routine(string);
114                     conn->number_of_live_retries = 0;
115                 }
116             }
117         }
118     }
119     CPU_ENABLE_INTERRUPTS;
120 }

```

000615



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

tasks.diags/hkeeping\_task.c

DATE

5/23/89

PAGE #

TIME

4:42:31 pm

2/2

LINE #

SOURCE TEXT

```
121
122 static void
123 set_test_led( )
124 {
125     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
126     p_cpu_ctl_reg->test_led = test_LED_ON,
127 }
128
129 static void
130 clear_test_led( )
131 {
132     cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,
133     p_cpu_ctl_reg->test_led = test_LED_OFF,
134 }
```

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM tasks.diags/receive_task.c	DATE	5/23/89	PAGE #
		TIME	4:42:31 pm	1/3

LINE #	SOURCE TEXT
1	/* SCCS ID: receive_task.c rev 3.1, 4/24/89 at 07:55:14 */
2	
3	#include <stdio.h>
4	#include "common.h"
5	#include "message.h"
6	#include "task.h"
7	#include "network.h"
8	
9	extern CONNECTION *table_of_conns[1];
10	
11	void
12	receive_task()
13	{
14	u_char user;
15	
16	if (init_socket() == FAILURE) {
17	printf("Unable to initialize socket\n");
18	dequeue_all_messages();
19	/* What should we do now? */
20	}
21	
22	for (;;) {
23	switch (lm_choose_connection(&user)) {
24	case PENDING:
25	if (lm_send_reply(table_of_conns[user]) != SUCCESS) {
26	printf("user id send reply failed\n", user);
27	break;
28	case FAILURE:
29	dequeue_all_messages();
30	break;
31	}
32	}
33	
34	static
35	dequeue_all_messages()
36	{
37	u_short type;
38	char str[MAX_MESSAGE];
39	
40	while (lm_dequeue_message(&type, str) != FAILURE) {
41	if (type == ERROR_MSG)
42	(void)printf("ERROR: %s\n", str);
43	else (void)printf("WARNING: %s\n", str);
44	}
45	
46	}
47	

Copyright 1989

SOURCE PROGRAM

DATE

5/23/89

PAGE #

Logic Modeling Systems

tasks.diaqs/serial\_task.c

TIME 4:42:31 pm

1/4

```

LINE # SOURCE TEXT
1  /* SOCS_ID: serial_task.c rev 3.1, 4/24/89 at 07:55:16 */
2  /*
3  ** serial_task
4  */
5  #include "fifo.h"
6  #include "task.h"
7  #include "common.h"
8  #include "getmsg.h"
9  #include "lm_diaqs.h"
10
11  static char Met_diag_message[256];
12
13  void
14  serial_task()
15  {
16      static char buffer;
17      struct fifo_entry fifo;
18
19      fifo.fifo_no = RX_FIFO;
20      fifo.user = 0;
21      fifo.task = SERIAL_TASK_ID;
22
23      for (;;) {
24          fifo.data = (char *)ac_getrc();
25          if (diag_fifo_put(&fifo) == FAILURE) printf("Fifo Failure\n");
26      }
27
28  void
29  modem_task()
30  {
31      static char buffer;
32      struct fifo_entry fifo;
33
34      fifo.fifo_no = RX_FIFO;
35      fifo.user = 1;
36      fifo.task = SERIAL_TASK_ID;
37
38      for (;;) {
39          fifo.data = (char *)getmodemc();
40          if (diag_fifo_put(&fifo) == FAILURE) printf("Fifo Failure\n");
41      }
42
43  send_error_message (conn, message)
44  {
45      CONNECTION *conn;
46      char *message;
47      long cmd;
48
49      cmd = LM_GET_LONG(conn);
50      output_error_message(conn, cmd, message);
51
52  output_error_message (conn, cmd, message)
53  {
54      CONNECTION *conn;
55      char *message;
56
57      register long i;
58
59      LM_CHK_PUT_LONG(conn, cmd + 1);
60      LM_CHK_PUT_LONG(conn, 1);
61      LM_CHK_PUT_CHAR(conn, (char) 1);
62      for (i = 0; i < strlen(message); i++)
63          LM_CHK_PUT_CHAR(conn, message[i]);
64      LM_CHK_PUT_CHAR(conn, 0);
65      lm_send_repl(conn);
66  }
67
68  char *
69  diag_get_mode(user, task)
70  {
71      if (task == RECEIVE_TASK_ID)
72          return Met_diag_message;
73      if (user == 0)
74          return "Modem is running diagnostics from the console\n";
75      else
76          return "Modem is running diagnostics over the modem\n";
77  }
78
79  #define LM_PEEK_LONG(X) \
80      (*((long *) (X) -> incoming_buffer_pointer))
81
82  diag_fifo_put(fifo)
83  {
84      struct fifo_entry *fifo;
85
86      static int User = 0;
87      static int Task = 0;
88      static char diag_username[32] = "unknown user";
89      static char diag_hostname[32] = "unknown host";
90      static char *mode;
91      char *s, *same_pointer;
92      CONNECTION *conn;
93      extern int cmd;
94      int cmd;
95
96      if (Task == 0) {
97          sprintf(Met_diag_message,
98              "%s is running diagnostics over the network from %s\n",
99              diag_username, diag_hostname);
100
101          if (fifo->task == RECEIVE_TASK_ID) {
102              conn = table_of_conns[fifo->user];
103              cmd = LM_PEEK_LONG(conn);
104              if (cmd != LM_DIAG_GINSE) {
105                  mode = "[RUNNING DIAGNOSTICS]\n";
106                  output_error_message(conn, cmd, mode);
107                  set_close_connection_for_server(conn);
108                  return SUCCESS;
109              }
110
111              if (4 < ((unsigned long *) (conn->incoming_buffer_pointer + 4)) {
112                  same_pointer = conn->incoming_buffer_pointer + 8;
113                  while (*s++ = *same_pointer++)
114                      ;
115                  s = diag_hostname;
116              }
117          }
118      }
119  }
120

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
tasks.diags/serial\_task.c

DATE 5/23/89  
TIME 4:42:31 pm

PAGE #  
2/5

LINE #	SOURCE TEXT
121	while (*p++ == "name_pointer")
122	{
123	
124	sprintf(net_diag_message,
125	"%s is running diagnostics over the network from %s\n",
126	diag_username, diag_hostname);
127	}
128	
129	User = fifo->user;
130	Task = fifo->task;
131	mode = diag_get_mode(User, Task);
132	printf(mode);
133	modemprintline(mode);
134	}
135	
136	if ((User == fifo->user) && (Task == fifo->task)) {
137	return fifo_put(fifo);
138	} else switch (fifo->task) {
139	case SERIAL_TASK_ID:
140	if ((char)fifo->data == INTR_CHARACTER) {
141	mode = diag_get_mode(fifo->user, fifo->task);
142	if (Task == RECEIVE_TASK_ID) {
143	/* close network connection */
144	conn = table_of_conns[User];
145	cmd = LM_PEEK_LONG(conn);
146	output_error_message(conn, cmd, mode);
147	}
148	User = fifo->user;
149	Task = fifo->task;
150	printf(mode);
151	modemprintline(mode);
152	return fifo_put(fifo);
153	} else {
154	if (fifo->user == 0) {
155	printf(mode);
156	} else {
157	modemprintline(mode);
158	}
159	}
160	break;
161	case RECEIVE_TASK_ID:
162	conn = table_of_conns[fifo->user];
163	cmd = LM_PEEK_LONG(conn);
164	output_error_message(conn, cmd, mode);
165	set_close_connection_for_server(conn);
166	break;
167	}
168	return SUCCESS;
169	}

1321

5,353,243

1322

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM tasks.diags/transmit_task.c		DATE 5/23/89	PAGE # 1/6
				TIME 4:42:31 pm	
LINE #	SOURCE TEXT				
1	/* SOCS_ID: transmit_task.c Rev: 1.1 4/24/89 at 07:55:19 */				
2	void				
3	transmit_task()				
4	{				
5	tx_task();				
6	}				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/delay.c

DATE 5/23/89  
TIME 6:14:35 pm

PAGE #  
1/1

```

LINE # SOURCE TEXT
1  /* SCSS_ID: delay.c rev 3.1, 4/24/89 at 07:52:37 */
2  #include "device.h"
3  #include "hardware.h"
4  #include "esprcm.h"
5  #include "lmsrver.h"
6  #include "protass.h"
7
8  get_delay(def_ptr, instance, ident_change, ident_inconsistent_pins,
9            source_pin_number, source_pin_value)
10 {
11     DEVICE_SPEC *def_ptr;
12     INSTANCE_INFO *instance;
13     PTRN_BITS_LONGWORD *ident_change;
14     PTRN_BITS_LONGWORD *ident_inconsistent_pins;
15     register u_short source_pin_number;
16     register u_short source_pin_value;
17
18     PIN_SPEC *pin_def;
19     DAB_INFO *dab_ptr;
20     PIN_INFO *pin_info;
21     MIN_TYP_MAX *min_ttyp_ptr;
22     MIN_TYP_MAX *old_ttyp_ptr;
23     register TIMING_SPEC *timing_ptr;
24     register u_short max_timing_ptr;
25     register u_short key_state;
26     register u_long ident_change_word;
27     register u_long mask;
28     register char bitno;
29     u_short dest_pin_number;
30     u_short unit_pin_number_offset;
31     u_short word_pin_number_offset;
32     u_char unitno;
33     u_char wordno;
34     dest_pin_value;
35
36     /* IF an output pin has several different path delays from the Eval/Store
37      * input pins, then the delay for that output is calculated as shown in the
38      * following table.
39      *
40      * PREVIOUS EVENT      CURRENT EVENT      ACTION
41      *
42      * none                eval table index == -1    table index == -1
43      * none                eval table index == x      table index == x
44      *
45      * table index == -1    eval table index == -1    table index == -1
46      * table index == x      eval table index == x      table index == x
47      * table index == -1    eval table index == -1    table index == x
48      * table index == x      eval table index == y      combine -> comp delay
49      * comp delay == x      eval table index == -1    ?no change
50      * comp delay == x      eval table index == x      combine
51      *
52      * none                store table index == -1    table index == -1
53      * none                store table index == x      table index == x
54      *
55      * table index == -1    store table index == -1    table index == -1
56      * table index == x      store table index == x      table index == x
57      * table index == -1    store table index == -1    table index == -1
58      * table index == x      store table index == x      table index == x
59      * comp delay == x      store table index == -1    table index == -1
60      * comp delay == x      store table index == x      table index == x
61      *
62      * NOTE:
63      * "none" -> this is the first event output
64      * "table index == -1" -> no delay is found in the delay table
65      * "comp delay" -> the delay is a combination of
66      *                 delay table entries
67      * "combine" -> combine the min/typ/max to get
68      *                 worst case delay
69      * "eval" -> this is an eval event
70      * "store" -> this is a store event
71
72     /* ??? Note: make sure that the control field of ident_change is 0.
73     */
74
75     dab_ptr = dab_list[instance->dab_info_index];
76     unit_pin_number_offset = 0;
77     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
78         word_pin_number_offset = unit_pin_number_offset + 79;
79         for (wordno = 0; wordno < 3; ++wordno) {
80             ident_change_word = ident_change[unitno].word[wordno];
81             for (bitno = 31; bitno >= 0; --bitno) {
82                 mask = bitno_to_mask(bitno);
83                 /* Get out of the "for" loop if there are no more set
84                  * bits to look at.
85                  */
86                 if (ident_change_word == 0)
87                     break;
88                 if (ident_change_word & mask) {
89                     /* reset the bit */
90                     ident_change_word = mask;
91                     dest_pin_number = word_pin_number_offset + bitno - 31;
92                     if (read_ptrn_bit_long(ident_inconsistent_pins[unitno],
93                                           wordno, (u_char)bitno) != 0) {
94                         dest_pin_value = LOGIC_U;
95                     }
96                     else
97                         dest_pin_value =
98                             read_pin_value_long(instance->last_sample_value,
99                                                    unitno, wordno, (u_char)bitno);
100                     pin_info = instance->pin_info_table[dest_pin_number];
101                     pin_def = ider_ptr->pin_table[dest_pin_number];
102                     timing_ptr = pin_def->delay_table[0];
103                     max_timing_ptr = timing_ptr->delay_cat;
104                     key_state = dest_pin_value << 4 | source_pin_value;
105                     for (; timing_ptr < max_timing_ptr; ++timing_ptr) {

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/delay.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

2/2

```

121          if (source_pin_number == timing_ptr->minor_pin &&
122              (u_short)(key_state & ((u_short *)timing_ptr)[1]) ==
123              key_state)
124              break;
125
126      pin_info->new_raw = dest_pin_value;
127      if (pin_info->output_pin_is_linked == FALSE) {
128          pin_info->output_pin_is_linked = TRUE;
129          pin_info->event_pin_number = source_pin_number;
130          pin_info->delay_type = DELAY_TABLE;
131          pin_info->next_output_pin_index =
132              instance->first_data_pin_index;
133          instance->first_data_pin_index = dest_pin_number;
134          if (timing_ptr == max_timing_ptr)
135              pin_info->min_delay = -1;
136          else
137              pin_info->min_delay = timing_ptr->ntyp_index;
138      }
139      else {
140          if (timing_ptr != max_timing_ptr) {
141              if (pin_info->delay_type == DELAY_TABLE) {
142                  if (pin_info->min_delay == -1) {
143                      pin_info->min_delay = timing_ptr->ntyp_index;
144                      pin_info->event_pin_number = source_pin_number;
145                      pin_info->delay_type = DELAY_TABLE;
146                  }
147                  else {
148                      new_ntyp_ptr = &def_ptr->ntyp_table[
149                          timing_ptr->ntyp_index];
150                      old_ntyp_ptr = &def_ptr->ntyp_table[
151                          pin_info->min_delay];
152                      combine_delay(&pin_info->min_delay,
153                                  &pin_info->typ_delay,
154                                  &pin_info->max_delay,
155                                  new_ntyp_ptr, old_ntyp_ptr);
156                      if (source_pin_number < pin_info->event_pin_number)
157                          pin_info->event_pin_number = source_pin_number;
158                      pin_info->delay_type = DELAY_TABLE_COMPOSITE;
159                  }
160              }
161              else {
162                  new_ntyp_ptr = &def_ptr->ntyp_table[
163                      timing_ptr->ntyp_index];
164                  old_ntyp_ptr = (MIN_TYP_MAX) &pin_info->min_delay;
165                  combine_delay(&pin_info->min_delay,
166                              &pin_info->typ_delay,
167                              &pin_info->max_delay,
168                              new_ntyp_ptr, old_ntyp_ptr);
169                  if (source_pin_number < pin_info->event_pin_number)
170                      pin_info->event_pin_number = source_pin_number;
171                  pin_info->delay_type = DELAY_TABLE_COMPOSITE;
172              }
173          }
174          word_pin_number_offset += 32;
175          unit_pin_number_offset += 80;
176      }
177
178      combine_delay(min, typ, max, delay1, delay2)
179      long
180      *min;
181      long
182      *typ;
183      long
184      *max;
185      MIN_TYP_MAX
186      *delay1;
187      MIN_TYP_MAX
188      *delay2;
189      {
190          if (delay1->minimum < delay2->minimum)
191              *min = delay1->minimum;
192          else
193              *min = delay2->minimum;
194          if (delay1->maximum > delay2->maximum)
195              *max = delay1->maximum;
196          else
197              *max = delay2->maximum;
198          *typ = (delay1->typical + delay2->typical) / 2;
199      }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/edgepl.c

DATE

5/23/89

PAGE #

TIME

6:14:35 pm

1/3

```

1  LINE # SOURCE TEXT
2  /* SCCS_ID: edgepl.c rev 3.1, 4/24/89 at 07:52:41 */
3  #include "device.h"
4  #include "hardware.h"
5  #include "common.h"
6  #include "lmerror.h"
7  #include "message.h"
8
9  #define NOT_FINISHED -1
10 #define FINISHED -2
11 #define THRESHOLD_FUDGE_FACTOR 4
12 #define def_ptr_def_duty_cycle 50
13
14 place_edges(def_ptr, dab_ptr)
15 DEVICE_SPEC *def_ptr;
16 DAB_INFO *dab_ptr;
17 {
18     EXTRA_DEVICE_SPEC *extra_def_ptr;
19     PIN_SPEC *pin_def;
20     u_long extra_time_needed;
21     u_long sample_min_delay;
22     u_long sample_jitter_error;
23     u_long early_sample_min_delay;
24     u_long early_sample_jitter_error;
25     u_long start_dead_time; /* dead time at the beginning of PCLK period */
26     u_long end_dead_time; /* dead time at the end of PCLK period */
27     u_long actual_end_dead_time;
28     u_long user_hold_time;
29     u_long user_setup_time;
30     u_long hold_time;
31     u_long setup_time;
32     u_long sample_edge_time1;
33     u_long sample_edge_time2;
34     u_long data_edge_time;
35     u_long store_edge_time;
36     u_long hsoff_edge_time;
37     u_long hsoff_edge_time2;
38     u_long clock_jitter_error;
39     u_long edge_jitter_error;
40     u_long quantization_error;
41     u_long physical_clock_period;
42     u_long logical_clock_period;
43     u_long xlogical_clock_period;
44     u_long min_logical_clock_period;
45     u_long max_logical_clock_period;
46     u_long max_modular_logical_clock_period;
47     u_long a_reg;
48     u_long b_reg;
49     u_long source_reg;
50     u_long temp;
51     u_long other_edge_time;
52     u_long constant_time;
53     u_long measured_period;
54     u_long edge_times(MAX_EDGE_COUNT);
55     u_long ret;
56     u_long minimum_fall_sample_time;
57     u_short pin_number;
58     u_char has_store_DWRZ_pins;
59     u_char has_store_R1_or_R2_pins;
60     u_char has_io_store_pins;
61     u_char bits_per_pin = 1; /* This will be declared in DAREDEF */
62     u_char loop_count = 0;
63     u_char all_io_store_pins_are_hard_driven;
64     u_char all_io_data_pins_are_hard_driven;
65     u_char any_hard_drives_io_pins;
66     u_char pattern_unit_count;
67     u_char threshold1;
68     u_char threshold2;
69     u_char threshold3;
70     u_char put_hsoff_edge_after_store_edge;
71
72     /* The declared clock_period in the DAREDEF is the device's clock period.
73      * If the device used DWRZ clock (but no R1/R2 clock) then this period
74      * must be divided by 2. Furthermore, if there are "n" unit-patterns/lane
75      * then this period is further divided by "n".
76      */
77
78     DPRINTF(("inside place_edges\n"));
79     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
80
81     all_io_store_pins_are_hard_driven = TRUE;
82     all_io_data_pins_are_hard_driven = TRUE;
83     has_store_DWRZ_pins = FALSE;
84     has_store_R1_or_R2_pins = FALSE;
85     has_io_store_pins = FALSE;
86     any_hard_drives_io_pins = FALSE;
87     put_hsoff_edge_after_store_edge = FALSE;
88
89     for (pin_number = 0; pin_number < def_ptr->pin_cnt; ++pin_number) {
90         pin_def = def_ptr->pin_table[pin_number];
91         if ((pin_def->direction == NONE) ||
92             (pin_def->direction == POWER) ||
93             (pin_def->direction == CROOEND) ||
94             (pin_def->direction == NC))
95             continue;
96
97         if (pin_def->pin_class == STORE) {
98             if (pin_def->clk_format == DWRZ)
99                 has_store_DWRZ_pins = TRUE;
100             else if ((pin_def->clk_format == R1) || (pin_def->clk_format == R2))
101                 has_store_R1_or_R2_pins = TRUE;
102         }
103
104         if (pin_def->direction == IO) {
105             if (pin_def->pin_class == STORE)
106                 has_io_store_pins = TRUE;
107
108             switch (pin_def->in_seq_drive) {
109                 case M_DRIVE:
110                 case NO_DRIVE:
111                     if (pin_def->pin_class == STORE)
112                         all_io_store_pins_are_hard_driven = FALSE;
113                     else
114                         all_io_data_pins_are_hard_driven = FALSE;
115                     break;
116                 case R_DRIVE:
117                 case R2_DRIVE:
118                     any_hard_drives_io_pins = TRUE;
119             }
120         }
121     }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

2/4

```

121         break;
122     default:
123         break;
124     }
125
126     switch (pin_def->last_cyc_drive) {
127     case R_DRIVE:
128     case RM_DRIVE:
129         any_hard_drives_to_pins = TRUE;
130         break;
131     default:
132         break;
133     }
134 }
135
136 pattern_unit_count = bits_per_pin * dab_ptr->unit_count_per_lane;
137
138 /* Measure the clock frequency if we are using external clock */
139 switch (def_ptr->clock_type) {
140 case INTERNAL:
141     /* We only multiply the hardware frequency by 2 if there are no R1/R2
142     * clocks because otherwise the R1/R2 clock will be running too fast.
143     */
144     if ((has_store_DWRZ_pins == TRUE) &&
145         (has_store_R1_or_R2_pins == FALSE)) {
146         min_logical_clock_period = def_ptr->clock_period1 / 2;
147         max_logical_clock_period = def_ptr->clock_period2 / 2;
148     }
149     else {
150         min_logical_clock_period = def_ptr->clock_period1;
151         max_logical_clock_period = def_ptr->clock_period2;
152     }
153
154     if (def_ptr->device_type == PRIVATE) {
155         def_ptr->clock_period1 = PRIVATE_PATTERN_PERIOD;
156         def_ptr->clock_period2 = MAX_MODELER_CLOCK_PERIOD;
157         min_logical_clock_period = PRIVATE_PATTERN_PERIOD;
158         max_logical_clock_period = MAX_MODELER_CLOCK_PERIOD;
159     }
160     break;
161 case EXTL:
162     source_reg = EXTL_REG_VALUE;
163
164     if (lm_tmg_measure_clock(EXTL_REG_VALUE, &measured_period) == FAILURE) {
165         lm_queue_message(ERROR_MSG, "Timing Generator error: measure clock");
166         return(FAILURE);
167     }
168
169     temp = measured_period * pattern_unit_count;
170
171     if ((has_store_DWRZ_pins == TRUE) &&
172         (has_store_R1_or_R2_pins == FALSE)) {
173         temp = temp * 2;
174     }
175
176     if (temp < def_ptr->clock_period1 * MAX_PERCENT_TOLERANCE / 100) {
177         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the maximum device_speed requirement (%0.2f MHz)",
178             (double)1000000.0 / (double)measured_period,
179             (double)1000000.0 / (double)def_ptr->clock_period1);
180         return(FAILURE);
181     }
182
183     if (temp > def_ptr->clock_period2 * MAX_PERCENT_TOLERANCE / 100) {
184         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the minimum device_speed requirement (%0.2f MHz)",
185             (double)1000000.0 / (double)measured_period,
186             (double)1000000.0 / (double)def_ptr->clock_period2);
187         return(FAILURE);
188     }
189
190     def_ptr->clock_period1 = temp;
191     def_ptr->clock_period2 = temp;
192
193     if ((has_store_DWRZ_pins == TRUE) &&
194         (has_store_R1_or_R2_pins == FALSE)) {
195         min_logical_clock_period = def_ptr->clock_period1 / 2;
196         max_logical_clock_period = def_ptr->clock_period2 / 2;
197     }
198     else {
199         min_logical_clock_period = def_ptr->clock_period1;
200         max_logical_clock_period = def_ptr->clock_period2;
201     }
202
203     break;
204 case EXTL2:
205     source_reg = EXTL2_REG_VALUE;
206
207     if (lm_tmg_measure_clock(EXTL2_REG_VALUE, &measured_period) == FAILURE) {
208         lm_queue_message(ERROR_MSG, "Timing Generator error: measure clock");
209         return(FAILURE);
210     }
211
212     temp = measured_period * pattern_unit_count;
213
214     if ((has_store_DWRZ_pins == TRUE) &&
215         (has_store_R1_or_R2_pins == FALSE)) {
216         temp = temp * 2;
217     }
218
219     if (temp < def_ptr->clock_period1 * MAX_PERCENT_TOLERANCE / 100) {
220         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the maximum device_speed requirement (%0.2f MHz)",
221             (double)1000000.0 / (double)measured_period,
222             (double)1000000.0 / (double)def_ptr->clock_period1);
223         return(FAILURE);
224     }
225
226     if (temp > def_ptr->clock_period2 * MAX_PERCENT_TOLERANCE / 100) {
227         lm_queue_message(ERROR_MSG, "The external clock frequency measured (%0.2f MHz) cannot satisfy the minimum device_speed requirement (%0.2f MHz)",
228             (double)1000000.0 / (double)measured_period,
229             (double)1000000.0 / (double)def_ptr->clock_period2);
230         return(FAILURE);
231     }
232
233     def_ptr->clock_period1 = temp;
234     def_ptr->clock_period2 = temp;
235
236     if ((has_store_DWRZ_pins == TRUE) &&
237         (has_store_R1_or_R2_pins == FALSE)) {
238         min_logical_clock_period = def_ptr->clock_period1 / 2;
239         max_logical_clock_period = def_ptr->clock_period2 / 2;
240     }
241     else {
242         min_logical_clock_period = def_ptr->clock_period1;
243         max_logical_clock_period = def_ptr->clock_period2;
244     }
245
246     break;
247 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

3/5

```

LINE # SOURCE TEXT
237 }
238
239 def_ptr->clock_period1 = temp;
240 def_ptr->clock_period2 = temp;
241
242 if ((has_store_EWE2_pins == TRUE) &&
243     (has_store_R1_or_R2_pins == FALSE)) {
244     min_logical_clock_period = def_ptr->clock_period1 / 2;
245     max_logical_clock_period = def_ptr->clock_period2 / 2;
246 }
247 else {
248     min_logical_clock_period = def_ptr->clock_period1;
249     max_logical_clock_period = def_ptr->clock_period2;
250 }
251
252 break;
253
254 default:
255     lm_queue_message(ERROR_MSG, "internal error: unknown clock type");
256     return(FAILURE);
257 }
258
259 /* Relax the frequency range by 1 % */
260 min_logical_clock_period = min_logical_clock_period / 100; /*
261
262 DPRINTF(("xedge_skew_time: %d\n", def_ptr->edge_skew_time));
263
264 /* Calculate HOLD time */
265 if (def_ptr->hold_time == -1)
266     hold_time = 15000;
267 else
268     hold_time = def_ptr->hold_time;
269
270 user_hold_time = hold_time;
271
272 DPRINTF(("xraw_hold_time: %d\n", hold_time));
273
274 hold_time += def_ptr->edge_skew_time;
275
276 if (all_io_store_pins_are_hard_drives == TRUE)
277     hold_time += def_ptr->hd_settling_time;
278 else
279     hold_time += def_ptr->md_settling_time;
280
281 DPRINTF(("xneeded_hold_time: %d\n", hold_time));
282
283 /* Calculate SETUP time */
284 if (def_ptr->setup_time == -1)
285     setup_time = 50000;
286 else
287     setup_time = def_ptr->setup_time;
288
289 user_setup_time = setup_time;
290
291 DPRINTF(("xraw_setup_time: %d\n", setup_time));
292
293 setup_time += def_ptr->edge_skew_time;
294
295 if (all_io_data_pins_are_hard_drives == TRUE)
296     setup_time += def_ptr->hd_settling_time;
297 else
298     setup_time += def_ptr->md_settling_time;
299
300 if (def_ptr->device_type == PUBLIC) {
301     if (any_hard_drives_io_pins == TRUE) {
302         /* ULTRA FAST */
303         if (has_io_store_pins == TRUE) {
304             if (setup_time < EDOFF_OFFSET + EDOFF_TO_STORE_OFFSET) {
305                 setup_time = EDOFF_OFFSET + EDOFF_TO_STORE_OFFSET;
306             }
307             if ((hold_time > EDOFF_HOLD_TIME_THRESHOLD) ||
308                 (max_logical_clock_period > EDOFF_PERIOD_THRESHOLD) ||
309                 (hold_time + setup_time > EDOFF_PERIOD_THRESHOLD)) {
310                 put_edoff_edge_after_store_edge = TRUE;
311             }
312         }
313         else {
314             if (setup_time < EDOFF_OFFSET)
315                 setup_time = EDOFF_OFFSET;
316         }
317     }
318 }
319
320 DPRINTF(("xneeded_setup_time: %d\n", setup_time));
321
322 if (min_logical_clock_period <
323     MIN_MODELER_CLOCK_PERIOD * pattern_unit_count) {
324     min_logical_clock_period = MIN_MODELER_CLOCK_PERIOD * pattern_unit_count;
325 }
326
327 if (def_ptr->clock_type == INTERNAL) {
328     /* Get the actual clock period achievable by the Timing Generator and
329     /* also the clock jitter error.
330     */
331     if (lm_tag_get_frequency_setting(min_logical_clock_period /
332                                     pattern_unit_count,
333                                     &physical_clock_period,
334                                     &clock_jitter_error,
335                                     &a_reg,
336                                     &b_reg,
337                                     &c_reg) == FAILURE) {
338         lm_queue_message(ERROR_MSG, "Timing Generator error: get frequency");
339         return(FAILURE);
340     }
341 }
342 else {
343     /* External clock --> clock jitter = 1% of period */
344     clock_jitter_error = measured_period / 100;
345     a_reg = 0;
346     b_reg = 0;
347     physical_clock_period = measured_period;
348 }
349
350 logical_clock_period = physical_clock_period * pattern_unit_count;
351
352 if (lm_tag_get_quantization_and_jitter_error(logical_clock_period,
353                                             &quantization_error,
354                                             &edge_jitter_error) == FAILURE) {
355     lm_queue_message(ERROR_MSG, "Timing Generator error: get quantization error");
356     return(FAILURE);
357 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

4/6

```

LINE # SOURCE TEXT
357
358 if (lm_tmg_get_dead_time(logical_clock_period,
359 start_dead_time,
360 end_dead_time) == FAILURE) {
361 lm_queue_message(ERROR_MSG, "Timing Generator error: get dead time");
362 return(FAILURE);
363 }
364
365 if (start_dead_time < MAGIC_OUTPUT_DEAD_TIME)
366 start_dead_time = MAGIC_OUTPUT_DEAD_TIME;
367
368 /* The Timing Generator can only generate edges on the first S_PCLK
369 periods of the logical clock period. If the number of pattern units
370 per line is greater than S, then consider the remaining PCLK periods
371 as end dead time in the edge time calculation.
372 */
373 if (pattern_unit_count > S) {
374 end_dead_time += (pattern_unit_count - S) * physical_clock_period;
375 }
376
377 if (def_ptr->device_type == PRIVATE) {
378 /* If PRIVATE mode, then increase the dead time by half of the clock
379 period to reserve place to put the DOFF edge.
380 */
381 actual_end_dead_time = end_dead_time;
382 end_dead_time += PRIVATE_PATTERN_PERIOD / 2;
383 }
384 else {
385 /* For PUBLIC device, increase the dead time by STORE_TO_EDOFF_OFFSET
386 * if put_half_edge_after_store_edge flag is TRUE.
387 */
388 if (put_half_edge_after_store_edge == TRUE) {
389 actual_end_dead_time = end_dead_time;
390 end_dead_time += STORE_TO_EDOFF_OFFSET + quantization_error +
391 2 * edge_jitter_error;
392 }
393 }
394
395 if (lm_tmg_get_sample_ramp_dead_time(logical_clock_period,
396 (u_long)EDGE75SAMPLETRIGGERMODE,
397 sample_min_delay,
398 sample_jitter_error) == FAILURE) {
399 lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
400 return(FAILURE);
401 }
402
403 if (lm_tmg_get_sample_ramp_dead_time(logical_clock_period,
404 (u_long)EARLYSAMPLETRIGGERMODE,
405 early_sample_min_delay,
406 early_sample_jitter_error) == FAILURE) {
407 lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
408 return(FAILURE);
409 }
410
411 max_modeler_logical_clock_period =
412 MAX_MODELER_CLOCK_PERIOD * pattern_unit_count;
413
414 DPRINTF(("clock jitter error: %d\n", clock_jitter_error));
415
416 while ((ret = calculate_edge_times(def_ptr,
417 logical_clock_period,
418 sample_min_delay, sample_jitter_error,
419 early_sample_min_delay,
420 edge_jitter_error, quantization_error,
421 start_dead_time, end_dead_time + clock_jitter_error,
422 hold_time + quantization_error + clock_jitter_error +
423 2 * edge_jitter_error,
424 setup_time + quantization_error + 2 * edge_jitter_error,
425 extra_time_needed,
426 sample_edge_time1,
427 sample_edge_time2,
428 data_edge_time,
429 store_edge_time) != FINISHED) {
430
431 if (ret == FAILURE)
432 return(FAILURE);
433
434 /* Return FAILURE if calculate_edge_times() returns NOT FINISHED */
435 if (def_ptr->clock_type != INTERNAL) {
436 lm_queue_message(ERROR_MSG, "the external clock period is too short to meet the setup and hold requirements");
437 return(FAILURE);
438 }
439
440 /* Save guard to avoid infinite loop */
441 --loop_count;
442 if (loop_count == 10) {
443 lm_queue_message(ERROR_MSG, "internal error: cannot place edges in 10 iterations");
444 return(FAILURE);
445 }
446
447 logical_clock_period += extra_time_needed;
448
449 if (logical_clock_period > max_modeler_logical_clock_period) {
450 lm_queue_message(ERROR_MSG, "this device requires clock period greater than maximum modeler clock period (%d ns)",
451 max_modeler_logical_clock_period);
452 return(FAILURE);
453 }
454
455 /* Note: different clock period might have different end_dead_time and
456 * start_dead_time because we might have to switch to different
457 * edge ramp.
458 */
459 if (lm_tmg_get_frequency_setting(logical_clock_period /
460 pattern_unit_count,
461 physical_clock_period,
462 clock_jitter_error,
463 lm_reg,
464 kh_reg,
465 (source_reg) == FAILURE) {
466 lm_queue_message(ERROR_MSG, "Timing Generator error: get frequency");
467 return(FAILURE);
468 }
469
470 DPRINTF(("clock jitter error: %d\n", clock_jitter_error));
471
472 logical_clock_period = physical_clock_period * pattern_unit_count;
473
474 if (lm_tmg_get_quantization_and_jitter_error(logical_clock_period,
475 quantization_error,
476 edge_jitter_error) == FAILURE) {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/edgepl.c

DATE	5/23/89	PAGE #
TIME	6:14:35 pm	5/7

```

LINE # SOURCE TEXT
477 lm_queue_message(ERROR_MSG, "Timing Generator error: get quantization error");
478 return(FAILURE);
479 }
480
481 if (lm_tmng_get_dead_time(logical_clock_period,
482                          actual_end_dead_time,
483                          end_dead_time) == FAILURE) {
484     lm_queue_message(ERROR_MSG, "Timing Generator error: get dead time");
485     return(FAILURE);
486 }
487
488 if (pattern_unit_count > 8) {
489     end_dead_time += (pattern_unit_count - 8) * physical_clock_period;
490 }
491
492 if (def_ptr->device_type == PRIVATE) {
493     /* If PRIVATE mode, then increase the dead time by half of the clock
494      * period to reserve place to put the DOFF edge.
495      */
496     actual_end_dead_time = end_dead_time;
497     end_dead_time += PRIVATE_PATTERN_PERIOD / 2;
498 }
499
500 else {
501     /* For PUBLIC device, increase the dead time by STORE_TO_HDOFF_OFFSET
502      * if put_hdooff_edge_after_store_edge flag is TRUE.
503      */
504     if (put_hdooff_edge_after_store_edge == TRUE) {
505         actual_end_dead_time = end_dead_time;
506         end_dead_time += STORE_TO_HDOFF_OFFSET + quantization_error +
507             2 * edge_jitter_error;
508     }
509 }
510
511 if (start_dead_time < MAGIC_OUTPUT_DEAD_TIME)
512     start_dead_time = MAGIC_OUTPUT_DEAD_TIME;
513
514 if (lm_tmng_get_sample_ramp_dead_time(logical_clock_period,
515                                       (ulong)EARLYSAMPLETRIGGERMODE,
516                                       sample_mis_delay,
517                                       sample_jitter_error) == FAILURE) {
518     lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
519     return(FAILURE);
520 }
521
522 if (lm_tmng_get_sample_ramp_dead_time(logical_clock_period,
523                                       (ulong)EARLYSAMPLETRIGGERMODE,
524                                       early_sample_mis_delay,
525                                       early_sample_jitter_error) == FAILURE) {
526     lm_queue_message(ERROR_MSG, "Timing Generator error: get sample ramp dead time");
527     return(FAILURE);
528 }
529
530 if (def_ptr->device_type == PUBLIC) {
531     if (any_hard_driver_is_plus == TRUE) {
532         /* Ultra Fast mode has DOFF edge HDOFF_OFFSET after data_edge_time */
533         hdooff_edge_time = data_edge_time + HDOFF_OFFSET +
534             quantization_error + 2 * edge_jitter_error;
535     }
536     else {
537         /* Non-Ultra Fast mode has DOFF edge at store_edge_time */
538         hdooff_edge_time = store_edge_time +
539             quantization_error + 2 * edge_jitter_error;
540     }
541     if (put_hdooff_edge_after_store_edge == TRUE) {
542         hdooff_edge_time2 = logical_clock_period - actual_end_dead_time;
543     }
544     else {
545         hdooff_edge_time2 = hdooff_edge_time;
546     }
547 }
548
549 else {
550     hdooff_edge_time = logical_clock_period - actual_end_dead_time;
551 }
552
553 /* It's OK to exceed the specified period by 1 %. This is necessary
554  * because the TMC can only return frequency with 1 % tolerance.
555  */
556 max_logical_clock_period += max_logical_clock_period / 100;
557
558 if (logical_clock_period > max_logical_clock_period) {
559     /* Adjust the logical_clock_period reported back to the user
560      * if necessary.
561      */
562     if ((has_store_DWRL_pins == TRUE) && (has_store_RL_OR_RE_pins == FALSE))
563         logical_clock_period = logical_clock_period * 2;
564     else
565         logical_clock_period = logical_clock_period;
566 }
567
568 if ((user_setup_time != 0) && (user_hold_time != 0)) {
569     lm_queue_message(WARNING_MSG, "device_name: %s being modeled with a device_speed of 0.2f MHz to guarantee td ss setup time and td ss
570     hold time; to increase the effective device_speed you must specify a smaller device_setup_time and/or a smaller device_hold_time.",
571                     def_ptr->device_name,
572                     (double)1000000.0 / ((double)logical_clock_period,
573                     user_setup_time / 1000,
574                     user_hold_time / 1000);
575 }
576
577 else if ((user_setup_time != 0) && (user_hold_time == 0)) {
578     lm_queue_message(WARNING_MSG, "device_name: %s being modeled with a device_speed of 0.2f MHz to guarantee td ss setup time and td ss
579     hold time; to increase the effective device_speed you must specify a smaller device_setup_time.",
580                     def_ptr->device_name,
581                     (double)1000000.0 / ((double)logical_clock_period,
582                     user_setup_time / 1000,
583                     user_hold_time / 1000);
584 }
585
586 else if ((user_setup_time == 0) && (user_hold_time != 0)) {
587     lm_queue_message(WARNING_MSG, "device_name: %s being modeled with a device_speed of 0.2f MHz to guarantee td ss setup time and td ss
588     hold time; to increase the effective device_speed you must specify a smaller device_hold_time.",
589                     def_ptr->device_name,
590                     (double)1000000.0 / ((double)logical_clock_period,
591                     user_setup_time / 1000,
592                     user_hold_time / 1000);
593 }
594
595 else if ((user_setup_time == 0) && (user_hold_time == 0)) {
596     lm_queue_message(WARNING_MSG, "device_name: %s being modeled with a device_speed of 0.2f MHz to guarantee td ss setup time and td ss
597     hold time; to increase the effective device_speed you must specify a smaller device_setup_time and/or a smaller device_hold_time.",
598                     def_ptr->device_name,
599                     (double)1000000.0 / ((double)logical_clock_period,
600                     user_setup_time / 1000,
601                     user_hold_time / 1000);
602 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/edgepl.c

DATE 5/23/89  
TIME 6:14:35 pm

PAGE #  
6/8

```

LINE #          SOURCE TEXT
593          user_hold_time / 1000);
594      }
595  }
596
597  extra_def_ptr->actua...clock_period = physical_clock_period;
598  extra_def_ptr->logica...clock_period = logical_clock_period;
599  extra_def_ptr->sample_edge_time1 = sample_edge_time1;
600  extra_def_ptr->sample_edge_time2 = sample_edge_time2;
601  extra_def_ptr->data_edge_time = data_edge_time;
602  extra_def_ptr->hdooff_edge_time = hdooff_edge_time;
603  extra_def_ptr->hdooff_edge_time2 = hdooff_edge_time2;
604  extra_def_ptr->store_edge_time = store_edge_time;
605  extra_def_ptr->s_reg = s_reg;
606  extra_def_ptr->k_reg = k_reg;
607  extra_def_ptr->source_reg = source_reg;
608
609  temp = extra_def_ptr->store_edge_time -
610        extra_def_ptr->logical_clock_period * def_ptr_def_duty_cycle / 100;
611
612  if ((long)temp < (long)start_dead_time) {
613      temp = start_dead_time;
614  }
615
616  extra_def_ptr->nea_significant_store_edge_time = temp;
617
618  /* Make sure that the hdooff edge is in the clock period */
619  if (def_ptr->device_type == PUBLIC) {
620      if (hdooff_edge_time > store_edge_time) {
621          hdooff_edge_time = store_edge_time;
622          lm_queue_message(ERROR_MSG, "internal error: HDOFF time > store time");
623          return(FAILURE);
624      }
625  }
626
627  edge_times[0] = hdooff_edge_time;
628  edge_times[1] = temp;
629  edge_times[2] = data_edge_time;
630  edge_times[3] = store_edge_time;
631  edge_times[4] = temp;
632  edge_times[5] = hdooff_edge_time2;
633
634  other_edge_time = temp;
635
636  center_time = temp + (store_edge_time - temp) / 2;
637
638  /* When using early sample for R1 or R2 clock, we don't want to
639   * search for the edge all the way back to the min threshold of the
640   * sample ramp because we might miss the pulse. This is due to the
641   * fact that the sample ramp is started before PClk. So we will only
642   * search back to the center of the pulse.
643   * This is not a problem with the late sample because the minimum
644   * threshold of the sample ramp will be just before the significant
645   * edge of the R1/R2 clock.
646   */
647  if (extra_def_ptr->store_event_mode == EARLYSAMPLETRIGGERMODE) {
648      if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, other_edge_time,
649          threshold1) == FAILURE) {
650          lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
651          return(FAILURE);
652      }
653      if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, center_time,
654          threshold2) == FAILURE) {
655          lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
656          return(FAILURE);
657      }
658      if (lm_tmg_get_early_sample_thres(RESOLUTION_05_NS, store_edge_time,
659          threshold3) == FAILURE) {
660          lm_queue_message(ERROR_MSG, "internal error: R1/R2 changes after the fastest sample ramp");
661          return(FAILURE);
662      }
663
664      DPRINTF(("leading center trailing: %d %d %d -> %d %d %d\n",
665          other_edge_time, center_time, store_edge_time,
666          threshold1, threshold2, threshold3));
667
668      if (threshold1 >= threshold2) {
669          lm_queue_message(ERROR_MSG, "internal error: center of pulse coincides with the leading edge of R1/R2 clock");
670          return(FAILURE);
671      }
672      if (threshold2 >= threshold3 - THRESHOLD_FUDGE_FACTOR) {
673          lm_queue_message(ERROR_MSG, "internal error: center of pulse coincides with the trailing edge of R1/R2 clock");
674          return(FAILURE);
675      }
676
677      extra_def_ptr->r1_or_r2_min_threshold = threshold2;
678      DPRINTF(("min threshold for R1 or R2 clock: %d\n", threshold2));
679  }
680
681  if (lm_tmg_get_edge_setting(logical_clock_period, edge_times,
682      extra_def_ptr->edge_setting) == FAILURE) {
683      lm_queue_message(ERROR_MSG, "Timing Generator error: get edge setting");
684      return(FAILURE);
685  }
686
687  if (extra_def_ptr->eval_event_mode == EARLYSAMPLETRIGGERMODE) {
688      extra_def_ptr->edge_setting[6] = 255;
689  }
690  else {
691      if (lm_tmg_get_sample_setting(logical_clock_period,
692          sample_edge_time1,
693          extra_def_ptr->edge_setting[6]) == FAILURE) {
694          lm_queue_message(ERROR_MSG, "Timing Generator error: get sample setting");
695          return(FAILURE);
696      }
697  }
698
699  if (extra_def_ptr->store_event_mode == EARLYSAMPLETRIGGERMODE) {
700      extra_def_ptr->edge_setting[7] = 255;
701  }
702  else {
703      if (lm_tmg_get_sample_setting(logical_clock_period,
704          sample_edge_time2,
705          extra_def_ptr->edge_setting[7]) == FAILURE) {
706          lm_queue_message(ERROR_MSG, "Timing Generator error: get sample setting");
707          return(FAILURE);
708      }
709  }
710
711  extra_def_ptr->edge_setting[8] = extra_def_ptr->edge_setting[6];
712

```

```

LINE # SOURCE TEXT
713
714 /* Set XDOFF edge to max(data_threshold + 1, threshold(datatime + 15)) */
715 if (extra_def_ptr->edge_setting[0] < extra_def_ptr->edge_setting[2] + 1) {
716     extra_def_ptr->edge_setting[0] = extra_def_ptr->edge_setting[2] + 1;
717     extra_def_ptr->edge_setting[5] = extra_def_ptr->edge_setting[2] + 1;
718 }
719
720 /* Make sure data edge and store edge are in different bucket. */
721 if (extra_def_ptr->edge_setting[2] == extra_def_ptr->edge_setting[3]) {
722     lm_queue_message(ERROR_MSG, "internal error: same threshold for data edge and store edge");
723     return(FAILURE);
724 }
725
726 /* Set the FALLING SAMPLE reg */
727 if (def_ptr->five_state_sample == 0) {
728     /* The user has not specified the five state sample in DANDEF,
729      * use the greater of (2*logical_clock_period) or 1.6 us.
730      * The 1.6 us sample time is OK whether or not we are doing
731      * timing measurement.
732      */
733     if (any_small_soft_driver(def_ptr) == TRUE) {
734         minimum_fall_sample_time = DEFAULT_FALL_SAMPLE_TIME2;
735     }
736     else {
737         minimum_fall_sample_time = DEFAULT_FALL_SAMPLE_TIME;
738     }
739 }
740
741 if (2 * logical_clock_period > minimum_fall_sample_time) {
742     /* Add 1 logical clock period to the sample time because
743      * the sample time is reference to the beginning of the last ptrn.
744      */
745     if (lm_tmg_get_falling_sample_setting(physical_clock_period,
746         (u_long)(3 * logical_clock_period),
747         &stamp) != SUCCESS) {
748         lm_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
749         return(FAILURE);
750     }
751     DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
752         physical_clock_period, 3 * logical_clock_period, stamp));
753 }
754 else {
755     /* Add 1 logical clock period to the sample time because
756      * the sample time is reference to the beginning of the last ptrn.
757      */
758     minimum_fall_sample_time += logical_clock_period;
759     if (lm_tmg_get_falling_sample_setting(physical_clock_period,
760         minimum_fall_sample_time,
761         &stamp) != SUCCESS) {
762         lm_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
763         return(FAILURE);
764     }
765     DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
766         physical_clock_period, minimum_fall_sample_time, stamp));
767 }
768
769 /* We are going to do timing measurement.
770 * The rising sample will be set at 255 * RESOLUTION_40_WS = 1000 ns.
771 * So the earliest time that the falling sample can happen is
772 * 1000 ns + MIN_SAMPLE_PULSE_WIDTH --> 1600 ns.
773 */
774 if (def_ptr->five_state_sample <
775     MAX_SAMPLE_TIME_RANGE * MIN_SAMPLE_PULSE_WIDTH) {
776     lm_queue_message(ERROR_MSG, "default sample time value too small; minimum value: %d ns",
777         (MAX_SAMPLE_TIME_RANGE * MIN_SAMPLE_PULSE_WIDTH) / 1000);
778     return(FAILURE);
779 }
780
781 /* Add 1 logical clock period to the sample time because
782 * the sample time is reference to the beginning of the last ptrn.
783 */
784 if (lm_tmg_get_falling_sample_setting(physical_clock_period,
785     def_ptr->five_state_sample + logical_clock_period,
786     &stamp) != SUCCESS) {
787     lm_queue_message(ERROR_MSG, "Timing Generator error: get falling sample setting");
788     return(FAILURE);
789 }
790
791 DPRINTF(("falling sample --> physical_clock_period: %d, delay: %d, setting: %d\n",
792     physical_clock_period,
793     def_ptr->five_state_sample + logical_clock_period, stamp));
794
795 extra_def_ptr->falling_sample_reg = stamp;
796
797 DPRINTF(("max_logical_period : %d\n", max_logical_clock_period));
798 DPRINTF(("min_logical_period : %d\n", min_logical_clock_period));
799 DPRINTF(("logical_clk_period : %d\n", logical_clock_period));
800 DPRINTF(("actual_phy_clk_period : %d\n", physical_clock_period));
801 DPRINTF(("sample_edge_eval : %d\n", sample_edge_time1));
802 DPRINTF(("sample_edge_store : %d\n", sample_edge_time2));
803 DPRINTF(("data_edge : %d\n", data_edge_time));
804 DPRINTF(("xhdoif_edge : %d\n", hdoif_edge_time));
805 DPRINTF(("xhdoif_edge2 : %d\n", hdoif_edge_time2));
806 DPRINTF(("xstore_edge : %d\n", store_edge_tj));
807 DPRINTF(("xother_edge : %d\n", other_edge_time));
808 DPRINTF(("xstart_dead_time : %d\n", start_dead_time));
809 DPRINTF(("xend_dead_time : %d\n", end_dead_time));
810 DPRINTF(("xsample_min_delay : %d\n", sample_min_delay));
811 DPRINTF(("xsample_jitter_error : %d\n", sample_jitter_error));
812 DPRINTF(("xearly_smp_min_delay : %d\n", early_sample_min_delay));
813 DPRINTF(("xearly_smp_jitter_error : %d\n", early_sample_jitter_error));
814 DPRINTF(("xedge_jitter_error : %d\n", edge_jitter_error));
815 DPRINTF(("xhdoif_time : %d\n", data_edge_time - logical_clock_period - store_edge_time));
816 DPRINTF(("xsetup_time : %d\n", store_edge_time - data_edge_time));
817 DPRINTF(("xquantization_error : %d\n", quantization_error));
818 DPRINTF(("xclock_jitter_error : %d\n", clock_jitter_error));
819 DPRINTF(("xn_reg : %d\n", a_reg));
820 DPRINTF(("xk_reg : %d\n", k_reg));
821 DPRINTF(("xsource_reg : %d\n", source_reg));
822 DPRINTF(("xedge_0_setting : %d\n", extra_def_ptr->edge_setting[0]));
823 DPRINTF(("xedge_1_setting : %d\n", extra_def_ptr->edge_setting[1]));
824 DPRINTF(("xedge_2_setting : %d\n", extra_def_ptr->edge_setting[2]));
825 DPRINTF(("xedge_3_setting : %d\n", extra_def_ptr->edge_setting[3]));

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/edgepl.c

DATE 5/23/89

PAGE #

TIME 6:14:35 pm

8/10

```

LINE # SOURCE TEXT
833 DPRINTF(("xEdge_4_setting : %d\n", extra_def_ptr->edge_setting(4)));
834 DPRINTF(("xEdge_5_setting : %d\n", extra_def_ptr->edge_setting(5)));
835 DPRINTF(("xEdge_6_setting : %d\n", extra_def_ptr->edge_setting(6)));
836 DPRINTF(("xEdge_7_setting : %d\n", extra_def_ptr->edge_setting(7)));
837 DPRINTF(("xfalling_sample_setting: %d\n",
838         extra_def_ptr->falling_sample_reg));
839 DPRINTF(("exiting place_edges\n"));
840 return(SUCCESS);
841 }
842
843
844 calculate_edge_times(def_ptr,
845                     clock_period,
846                     sample_min_delay, sample_jitter_error,
847                     early_sample_min_delay,
848                     edge_jitter_error, quantization_error,
849                     start_pclk_dead_time, end_pclk_dead_time,
850                     hold_time, setup_time,
851                     extra_time_needed,
852                     sample_edge_time1, sample_edge_time2,
853                     data_edge_time, store_edge_time)
854
855 DEVICE_SPEC *def_ptr,
856 u_long clock_period,
857 u_long sample_min_delay,
858 u_long sample_jitter_error,
859 u_long early_sample_min_delay,
860 u_long edge_jitter_error,
861 u_long quantization_error,
862 u_long start_pclk_dead_time,
863 u_long end_pclk_dead_time,
864 u_long hold_time,
865 u_long setup_time,
866 u_long extra_time_needed,
867 u_long sample_edge_time1,
868 u_long sample_edge_time2,
869 u_long data_edge_time,
870 u_long store_edge_time;
871
872 EXTRA_DEVICE_SPEC *extra_def_ptr,
873 u_long store_edge,
874 u_long data_edge,
875 u_long current_setup_time,
876 u_long sample_ramp_dead_time,
877 u_long edge7_sample_ramp_dead_time,
878
879 DPRINTF(("inside calculate_edge_times\n"));
880 DPRINTF(("clock period : %d\n", clock_period));
881 DPRINTF(("sample_min_delay : %d\n", sample_min_delay));
882 DPRINTF(("sample_jitter_error : %d\n", sample_jitter_error));
883 DPRINTF(("early_sample_min_delay : %d\n", early_sample_min_delay));
884 DPRINTF(("edge_jitter_error : %d\n", edge_jitter_error));
885 DPRINTF(("quantization_error : %d\n", quantization_error));
886 DPRINTF(("start_pclk_dead_time : %d\n", start_pclk_dead_time));
887 DPRINTF(("end_pclk_dead_time : %d\n", end_pclk_dead_time));
888 DPRINTF(("hold_time : %d\n", hold_time));
889 DPRINTF(("setup_time : %d\n", setup_time));
890
891 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data,
892
893 edge7_sample_ramp_dead_time = sample_min_delay + sample_jitter_error +
894                               edge_jitter_error + quantization_error,
895
896 if (early_sample_min_delay < sample_min_delay) {
897     /* use EARLY SAMPLE mode */
898     sample_ramp_dead_time = early_sample_min_delay,
899 }
900 else {
901     /* use EDGE 7 SAMPLE mode */
902     sample_ramp_dead_time = sample_min_delay + sample_jitter_error +
903                             edge_jitter_error + quantization_error,
904 }
905
906 if (extra_def_ptr->has_io_store == TRUE) {
907     /* IO store pin can change to 1 at PCLK time, so the hold time
908      * has to be satisfied before the next PCLK.
909     */
910     store_edge = - hold_time,
911
912     if (store_edge > - (long)end_pclk_dead_time)
913         store_edge = - end_pclk_dead_time,
914
915     if (start_pclk_dead_time < sample_ramp_dead_time)
916         data_edge = sample_ramp_dead_time,
917     else
918         data_edge = start_pclk_dead_time,
919 }
920 else {
921     /* Does NOT have I/O store */
922     store_edge = -end_pclk_dead_time,
923
924     data_edge = store_edge + hold_time,
925
926     if (start_pclk_dead_time < sample_ramp_dead_time) {
927         /* sample_ramp_dead_time is the limiting factor */
928         if (data_edge < (long)sample_ramp_dead_time)
929             data_edge = sample_ramp_dead_time,
930     }
931     else {
932         /* start_pclk_dead_time is the limiting factor */
933         if (data_edge < (long)start_pclk_dead_time)
934             data_edge = start_pclk_dead_time,
935     }
936 }
937
938 *store_edge_time = clock_period + store_edge,
939 *data_edge_time = data_edge,
940
941 *sample_edge_time1 = sample_min_delay,
942 *sample_edge_time2 = sample_min_delay,
943
944 if (*data_edge_time >= edge7_sample_ramp_dead_time) {
945     /* Use EDGE7 SAMPLE for EVAL event */
946     extra_def_ptr->eval_event_mode = EDGE7/SAMPLETRIGGERMODE,
947     *sample_edge_time1 = *data_edge_time -
948                         (sample_jitter_error + edge_jitter_error + quantization_error),
949     DPRINTF(("Use edge7 sample for EVAL event\n"));
950 }
951 else {
952     /* Use EARLY SAMPLE for EVAL event */

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/edgepl.c

DATE 5/23/89  
TIME 6:14:35 pm

PAGE #  
9/11

```

LINE #          SOURCE TEXT
953      extra_def_ptr->eval_event_mode = EARLYSAMPLETRIGGERMODE;
954      DPRINTF(("use early sample for EVAL event\n"));
955      }
956
957      if (*store_edge_time >= edge7_sample_ramp_dead_time) {
958          /* Use EDGE7 SAMPLE for STORE event */
959          extra_def_ptr->store_event_mode = EDGE7SAMPLETRIGGERMODE;
960          *sample_edge_time2 = *store_edge_time -
961              (sample_jitter_error + edge_jitter_error + quantization_error);
962          DPRINTF(("use edge7 sample for STORE event\n"));
963      }
964      else {
965          /* Use EARLY SAMPLE for STORE event */
966          extra_def_ptr->store_event_mode = EARLYSAMPLETRIGGERMODE;
967          DPRINTF(("use early sample for STORE event\n"));
968      }
969
970      current_setup_time = *store_edge_time - *data_edge_time;
971
972      DPRINTF(("store edge time      : %d\n", *store_edge_time));
973      DPRINTF(("data edge time2      : %d\n", *data_edge_time));
974      DPRINTF(("sample edge time1 (eval): %d\n", *sample_edge_time1));
975      DPRINTF(("sample edge time2 (store): %d\n", *sample_edge_time2));
976
977      if (current_setup_time < (long)setup_time) {
978          *extra_time_needed = setup_time - current_setup_time;
979          DPRINTF(("extra time      : %d\n", *extra_time_needed));
980          return(NOT_FINISHED);
981      }
982
983      return(FINISHED);
984      }
985
986      any_small_soft_driver(def_ptr)
987      DEVICE_SPEC *def_ptr;
988      {
989          /* If only the weakest soft drive high or low is on then return TRUE */
990
991          PIN_SPEC    *pin_spec_ptr;
992          u_short      pin_count;
993          u_short      pinno;
994
995          pin_count = def_ptr->pin_cnt;
996          pin_spec_ptr = &def_ptr->pin_table[0];
997          for (pinno = 0; pinno < pin_count; ++pinno) {
998              if (pin_spec_ptr->direction != IO) {
999                  ++pin_spec_ptr;
1000                  continue;
1001              }
1002
1003              if ((pin_spec_ptr->s_drive_low == 1) ||
1004                  (pin_spec_ptr->s_drive_hi == 1))
1005                  return(TRUE);
1006
1007              ++pin_spec_ptr;
1008          }
1009
1010          return(FALSE);
1011      }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/fault.c

DATE 5/23/89  
TIME 6:14:37 pm

PAGE #  
1/12

```

1  /* SCCS_ID: fault.c rev 3.1, 4/24/89 at 07:52:46 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "eeprom.h"
7  #include "lmserver.h"
8
9  setup_fault_pattern(user, inst_ptr, table_index)
10 USER_INFO user;
11 INSTANCE_INFO inst_ptr;
12 u_short table_index;
13 {
14     DEVICE_SPEC dev_ptr;
15     DAB_INFO dab_ptr;
16     INSTANCE_INFO inst_ptr;
17     LANE_ADDR_INFO lane_info;
18     u_long temp_inst_unit_addr;
19     u_long temp_fault_unit_addr;
20     u_short fault_id_table_index;
21     u_short total_unit;
22     u_char lane;
23     u_char unitno;
24     u_short prev_block_number;
25     u_long block_number;
26     u_long link_table_addr;
27     u_short next_to_last_block_number;
28     u_char temp;
29
30     dev_ptr = inst_ptr->dev_info;
31
32     dab_ptr = dab_list(inst_ptr->dab_info_index);
33
34     total_unit = dab_ptr->unit_count_per_lane * dab_ptr->lane_count;
35
36     if (new_and_link_instance(user, dev_ptr, "fault",
37         dab_ptr->unit_count,
38         fault_id_table_index,
39         (char)inst_ptr->dab_info_index) == FAILURE) {
40         in_queue_message(ERROR_MSG, "out of memory on modeler for instance");
41         return(FAILURE);
42     }
43
44     fault_ptr = user->instances[fault_id_table_index];
45
46     fault_ptr->is_fault = TRUE;
47
48     copy_instance_info(inst_ptr, fault_ptr);
49
50     fault_ptr->disjoint_flag = FALSE;
51
52     /* allocate new block in each lane and store address in VAR_SEQ_ADDR */
53     for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
54         if (dab_ptr->lane_used[lane]) {
55             if (new_block(&fault_ptr->var_seq_addr[lane], &temp,
56                 PATTERN_BLOCK, lane) == FAILURE) {
57                 fault_ptr->failed_to_alloc_ptr = TRUE;
58                 in_queue_message(ERROR_MSG, "out of Fast Pattern Memory");
59                 return(FAILURE);
60             }
61         }
62     }
63
64     if (last_pattern_spans_2_blocks(inst_ptr) == TRUE) {
65         /* If the last pattern of the instance spans 2 blocks, this means:
66          * that the last 2 blocks of the instance are NOT feedback blocks.
67          * It also means that the instance has at least 2 blocks.
68          */
69
70         /* find the address of the last common block from the beginning of
71          * the instance pattern sequence.
72          */
73         for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
74             if (dab_ptr->lane_used[lane]) {
75                 prev_block_number = 0;
76                 block_number = ptob(inst_ptr->seq_start_addr[lane]);
77                 link_table_addr = ptol(inst_ptr->seq_start_addr[lane]);
78
79                 next_to_last_block_number =
80                     ptob(inst_ptr->lane_addr[lane].prev_max_addr - PTRN_ADDR_INC);
81
82                 while (block_number != next_to_last_block_number) {
83                     prev_block_number = block_number;
84                     block_number = read_loc_long((u_long *)link_table_addr) &
85                         BLOCK_NUMBER_MASK;
86                     link_table_addr = btol(lane, block_number);
87                 }
88
89                 if (prev_block_number == 0) {
90                     /* There are only 2 blocks in the instance sequence, so
91                      * the fault sequence has to be disjoint from the instance seq.
92                      * The condition will never happen when the instance has
93                      * feedback sequence, because there would be at least 4 blocks
94                      * in the instance sequence:
95                      * 1- block PREPARE + pre feedback seq
96                      * 2- blocks feedback seq
97                      * 2- blocks post feedback seq + real user pattern seq
98                      * (at least 2 blocks because the last pattern
99                       * spans 2 blocks)
100
101                     Therefore we will never have to worry about allocating
102                     feedback blocks when we are creating faults.
103
104                     fault_ptr->disjoint_flag = TRUE;
105
106                     fault_ptr->seq_start_addr[lane] =
107                         fault_ptr->var_seq_addr[lane];
108
109                     fault_ptr->last_common_block_addr[lane] = 0;
110
111                     fault_ptr->last_common_block_is_feedback = FALSE;
112                 }
113             }
114             else {
115                 fault_ptr->seq_start_addr[lane] =
116                     inst_ptr->seq_start_addr[lane];
117
118                 fault_ptr->last_common_block_addr[lane] =
119                     btol(lane, prev_block_number);
120             }
121         }
122     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/fault.c

DATE 5/23/89  
TIME 6:14:37 pm

PAGE #  
2/13

```

LINE # SOURCE TEXT
121 if (fault_ptr->last_common_block_addr[laneso] ==
122     inst_ptr->fb_block_addr[laneso]) {
123     fault_ptr->last_common_block_is_feedback = TRUE;
124 }
125 else {
126     fault_ptr->last_common_block_is_feedback = FALSE;
127 }
128
129
130
131 /* copy patterns from the instance next to last block to the
132    * fault block addressed by VAR_SEQ_ADDR.
133 */
134 copy_patterns(inst_ptr->lane_addr[laneso].prev_max_addr -
135             BLOCK_ADDR_INC,
136             fault_ptr->var_seq_addr[laneso],
137             PTRN_PER_BLOCK);
138
139 lane_info = &fault_ptr->lane_addr[laneso];
140
141 lane_info->prev_max_addr =
142     fault_ptr->var_seq_addr[laneso] + BLOCK_ADDR_INC;
143
144 if (new_block(lane_info->max_addr,
145             &temp,
146             PATTERN_BLOCK,
147             laneso) == FAILURE) {
148     lm_printf_message(ERROR_MSG, "out of Fast Pattern Memory");
149     fault_ptr->failed_to_alloc_ptrn = TRUE;
150     return(FAILURE);
151 }
152
153 set_branch(lane_info->prev_max_addr - PTRN_ADDR_INC,
154            lane_info->max_addr);
155
156 lane_info->max_addr += BLOCK_ADDR_INC;
157
158 copy_patterns(inst_ptr->lane_addr[laneso].max_addr -
159             BLOCK_ADDR_INC,
160             fault_ptr->lane_addr[laneso].max_addr -
161             BLOCK_ADDR_INC,
162             PTRN_PER_BLOCK);
163 }
164
165
166 temp_inst_unit_addr = &inst_ptr->
167     unit_addr[inst_ptr->cur_unit_addr_index][0];
168
169 temp_fault_unit_addr = &fault_ptr->
170     unit_addr[fault_ptr->cur_unit_addr_index][0];
171
172 for (unitno = 0; unitno < total_unit; ++unitno) {
173     laneso = dab_ptr->unit_location[unitno].lane_no;
174
175     /* Check if the unit is in the next to last block */
176     if ((temp_inst_unit_addr[unitno] ==
177         (inst_ptr->lane_addr[laneso].prev_max_addr - BLOCK_ADDR_INC)) &&
178         (temp_inst_unit_addr[unitno] <
179         (inst_ptr->lane_addr[laneso].prev_max_addr))) {
180
181         temp_fault_unit_addr[unitno] =
182             fault_ptr->lane_addr[laneso].prev_max_addr -
183             (inst_ptr->lane_addr[laneso].prev_max_addr -
184             temp_inst_unit_addr[unitno]);
185     }
186     else {
187         temp_fault_unit_addr[unitno] =
188             fault_ptr->lane_addr[laneso].max_addr -
189             (inst_ptr->lane_addr[laneso].max_addr -
190             temp_inst_unit_addr[unitno]);
191     }
192
193     fault_ptr->first_user_ptrn_unit_addr[unitno] =
194         temp_fault_unit_addr[unitno];
195
196     if (dab_ptr->unit_location[unitno].last_in_lane)
197         fault_ptr->lane_addr[laneso].last_unit_addr =
198             temp_fault_unit_addr[unitno];
199
200     fault_ptr->lane_addr[laneso].new_block_addr = 0;
201 }
202
203 }
204
205 else {
206     /* The last pattern of the instance is completely contained in the
207        * the last instance block.
208        */
209     for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
210         if (!dab_ptr->lane_used[laneso])
211             continue;
212
213         if (inst_ptr->lane_addr[laneso].prev_max_addr == 0) {
214             /* The instance only has 1 block */
215
216             fault_ptr->disjoint_flag = TRUE;
217
218             fault_ptr->seq_start_addr[laneso] =
219                 fault_ptr->var_seq_addr[laneso];
220
221             fault_ptr->last_common_block_addr[laneso] = 0;
222         }
223         else {
224             fault_ptr->seq_start_addr[laneso] =
225                 inst_ptr->seq_start_addr[laneso];
226
227             /* IF there are feedback sequence, PREV_MAX_ADDR will point
228                * to the end of the feedback blocks. We want to make
229                * LAST COMMON BLOCK_ADDR point to the beginning of the
230                * feedback blocks.
231                */
232             if (inst_ptr->lane_addr[laneso].prev_max_addr ==
233                 (inst_ptr->fb_block_size[laneso] +
234                 BLOCK_ADDR_INC)) {
235                 /* PREV_MAX_ADDR points to the end of feedback blocks */
236                 fault_ptr->last_common_block_addr[laneso] =
237                     inst_ptr->fb_block_addr[laneso];
238             }
239         }
240     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/fault.c

DATE	5/23/89	PAGE #
TIME	6:14:37 pm	3/14

```

LINE #      SOURCE TEXT
241      fault_ptr->last_common_block_is_feedback = TRUE;
242      }
243      else {
244      fault_ptr->last_common_block_addr[common] =
245      inst_ptr->lane_addr[lane].prev_max_addr - BLOCK_ADDR_INC,
246      fault_ptr->var_seq_addr[lane],
247      PTRN_PTR_BLOCK);
248      fault_ptr->last_common_block_is_feedback = FALSE;
249      }
250
251      copy_pattern(inst_ptr->lane_addr[lane].max_addr -
252      BLOCK_ADDR_INC,
253      fault_ptr->var_seq_addr[lane],
254      PTRN_PTR_BLOCK);
255
256      /* This is necessary to set the SEQ_END bit */
257      fault_ptr->lane_addr[lane].prev_max_addr =
258      inst_ptr->lane_addr[lane].prev_max_addr;
259
260      fault_ptr->lane_addr[lane].max_addr =
261      fault_ptr->var_seq_addr[lane] + BLOCK_ADDR_INC;
262      }
263
264      temp_inst_unit_addr = inst_ptr->
265      unit_addr(inst_ptr->cur_unit_addr_index)[0];
266
267      temp_fault_unit_addr = fault_ptr->
268      unit_addr(fault_ptr->cur_unit_addr_index)[0];
269
270      /* Setup the unit address for the fault */
271      for (unitno = 0; unitno < total_unit; ++unitno) {
272
273      lane = dab_ptr->unit_location[unitno].lane_no;
274
275      temp_fault_unit_addr[unitno] =
276      fault_ptr->lane_addr[lane].max_addr -
277      (inst_ptr->lane_addr[lane].max_addr -
278      temp_inst_unit_addr[unitno]);
279
280      fault_ptr->first_user_ptrn_unit_addr[unitno] =
281      temp_fault_unit_addr[unitno];
282
283      if (dab_ptr->unit_location[unitno].last_in_lane)
284      fault_ptr->lane_addr[lane].last_unit_addr =
285      temp_fault_unit_addr[unitno];
286
287      fault_ptr->lane_addr[lane].new_block_addr = 0;
288      }
289
290      }
291
292      /* Setup the LCHDB and LCHDB addresses.
293      * If the fault is not disjoint from the instance, then
294      * just copy the lchdb and lchdb addresses from the instance.
295      * If the fault is disjoint from the instance, then we can see above
296      * that the instance can have at most 2 blocks.
297      */
298
299      if (fault_ptr->disjoint_flag == FALSE) {
300      for (unitno = 0; unitno < total_unit; ++unitno) {
301      fault_ptr->lchdb_addr[unitno] =
302      inst_ptr->lchdb_addr[unitno];
303
304      fault_ptr->lchdb_addr[unitno] =
305      inst_ptr->lchdb_addr[unitno];
306      }
307      }
308      else {
309      for (unitno = 0; unitno < total_unit; ++unitno) {
310      lane = dab_ptr->unit_location[unitno].lane_no;
311
312      /* Check if the instance's lchdb is in prev block */
313      if ((inst_ptr->lchdb_addr[unitno] >=
314      (inst_ptr->lane_addr[lane].prev_max_addr - BLOCK_ADDR_INC)) &&
315      (inst_ptr->lchdb_addr[unitno] <
316      (inst_ptr->lane_addr[lane].prev_max_addr))) {
317      fault_ptr->lchdb_addr[unitno] =
318      fault_ptr->lane_addr[lane].prev_max_addr -
319      (inst_ptr->lane_addr[lane].prev_max_addr -
320      inst_ptr->lchdb_addr[unitno]);
321      }
322      else {
323      fault_ptr->lchdb_addr[unitno] =
324      fault_ptr->lane_addr[lane].max_addr -
325      (inst_ptr->lane_addr[lane].max_addr -
326      inst_ptr->lchdb_addr[unitno]);
327      }
328
329      /* Check if the instance's lchdb is in prev block */
330      if ((inst_ptr->lchdb_addr[unitno] >=
331      (inst_ptr->lane_addr[lane].prev_max_addr - BLOCK_ADDR_INC)) &&
332      (inst_ptr->lchdb_addr[unitno] <
333      (inst_ptr->lane_addr[lane].prev_max_addr))) {
334      fault_ptr->lchdb_addr[unitno] =
335      fault_ptr->lane_addr[lane].prev_max_addr -
336      (inst_ptr->lane_addr[lane].prev_max_addr -
337      inst_ptr->lchdb_addr[unitno]);
338      }
339      else {
340      fault_ptr->lchdb_addr[unitno] =
341      fault_ptr->lane_addr[lane].max_addr -
342      (inst_ptr->lane_addr[lane].max_addr -
343      inst_ptr->lchdb_addr[unitno]);
344      }
345      }
346      }
347
348      *table_index = fault_id_table_index;
349
350      return(SUCCESS);
351
352      }
353
354      modify_instance_pattern_addr(inst_ptr, common_ptrn_count,
355      pattern_count,
356      unit_addr,
357      unit_addr2,
358      max_addr,
359      prev_max_addr)
360

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/fault.c

DATE 5/23/89  
TIME 6:14:37 pm

PAGE #  
4/15

```

LINE # SOURCE TEXT
361 INSTANCE_INFO *inst_ptr;
362 long common_ptr_count;
363 u_long *pattern_count;
364 u_long unit_addr[];
365 u_long unit_addr2[];
366 u_long max_addr[];
367 u_long prev_max_addr[];
368 {
369 /* This routine changes the following information from the instance which
370 * which is used by setup_fault_pattern() and saves the information in
371 * the temporary variables:
372 * pattern count
373 * unit_addr[0]
374 * unit_addr[1]
375 * prev_max_addr
376 * max_addr
377 */
378
379 DAB_INFO *dab_ptr;
380 u_long max_unit_addr[MAX_LANE_COUNT];
381 u_long link_table_addr;
382 u_long prev_block_number;
383 u_long cur_block_number;
384 u_long block_number;
385 u_long *temp_unit_addr;
386 u_long *temp_unit_addr2;
387 u_char laneso;
388 u_char unitno;
389
390 dab_ptr = dab_list(inst_ptr->dab_info_index);
391
392 /* Save/modify pattern count */
393 pattern_count = inst_ptr->pattern_count;
394 inst_ptr->pattern_count = inst_ptr->static_pattern_count +
395 dab_ptr->unit_count_per_lane;
396
397 /* Save instance's unit_addr and previous unit_addr */
398 temp_unit_addr = inst_ptr->unit_addr[inst_ptr->cur_unit_addr_index][0];
399 temp_unit_addr2 =
400 inst_ptr->unit_addr[inst_ptr->cur_unit_addr_index + 1 & 1][0];
401
402 for (unitno = 0; unitno < MAX_UNIT_COUNT; ++unitno) {
403
404     unit_addr[unitno] = temp_unit_addr[unitno];
405     unit_addr2[unitno] = temp_unit_addr2[unitno];
406
407     temp_unit_addr[unitno] =
408     inst_ptr->first_user_ptr_unit_addr[unitno];
409 }
410
411 /* Save the max_addr, prev_max_addr */
412 for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
413
414     max_addr[laneso] = inst_ptr->lane_addr[laneso].max_addr;
415     prev_max_addr[laneso] = inst_ptr->lane_addr[laneso].prev_max_addr;
416     max_unit_addr[laneso] = 0;
417 }
418
419 while (common_ptr_count > 0) {
420
421     increment_unit_addr(temp_unit_addr,
422 dab_ptr->unit_count,
423 dab_ptr->unit_count_per_lane);
424
425     inst_ptr->pattern_count += dab_ptr->unit_count_per_lane;
426     --common_ptr_count;
427 }
428
429 /* find the maximum unit_addr */
430 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
431     laneso = dab_ptr->unit_location[unitno].lane_no;
432
433     if (max_unit_addr[laneso] < temp_unit_addr[unitno])
434         max_unit_addr[laneso] = temp_unit_addr[unitno];
435 }
436
437 /* modify the max_addr, prev_max_addr */
438 for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
439
440     if (!dab_ptr->lane_used[laneso])
441         continue;
442
443     inst_ptr->lane_addr[laneso].max_addr =
444     (max_unit_addr[laneso] & BLOCK_START_MASK) + BLOCK_ADDR_INC;
445
446     /* search the block before the current block occupied by the last unit */
447     prev_block_number = 0;
448     block_number = ptob(inst_ptr->seq_start_addr[laneso]);
449     link_table_addr = ptob(inst_ptr->seq_start_addr[laneso]);
450     cur_block_number = ptob(max_unit_addr[laneso]);
451
452     while (block_number != cur_block_number) {
453
454         prev_block_number = block_number;
455         block_number = read_loc_long((u_long *)link_table_addr) &
456             BLOCK_NUMBER_MASK;
457         link_table_addr = btol(laneso, block_number);
458     }
459
460     if (prev_block_number == 0)
461         inst_ptr->lane_addr[laneso].prev_max_addr = 0;
462     else {
463         inst_ptr->lane_addr[laneso].prev_max_addr =
464         btob(laneso, prev_block_number) + BLOCK_ADDR_INC;
465
466         if ((inst_ptr->lane_addr[laneso].prev_max_addr - BLOCK_ADDR_INC) ==
467             inst_ptr->fb_block_addr[laneso])
468             inst_ptr->lane_addr[laneso].prev_max_addr +=
469             (inst_ptr->fb_block_size[laneso] - 1) * BLOCK_ADDR_INC;
470     }
471 }
472
473
474
475
476
477
478
479
480

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/fault.c

DATE 5/23/89 PAGE #  
TIME 6:14:37 pm 5/16

```

LINE # SOURCE TEXT
481 }
482
483 restore_instance_patterns_addr(inst_ptr,
484                               patterns_count,
485                               unit_addr,
486                               unit_addr2,
487                               max_addr,
488                               prev_max_addr)
489
490 INSTANCE_INFO *inst_ptr,
491 u_long patterns_count,
492 u_long unit_addr[1],
493 u_long max_addr[1],
494 u_long prev_max_addr[1],
495
496 /* This routine restores the following information for the instance which
497  * which was saved in the temporary variables:
498  * patterns_count
499  * unit_addr[0]
500  * unit_addr[1]
501  * prev_max_addr
502  * max_addr
503  */
504
505 u_long temp_unit_addr,
506 u_char laneso,
507 u_char unitno,
508
509
510 /* Restore patterns_count */
511 inst_ptr->patterns_count = patterns_count,
512
513 /* Restore unit_addr[0] */
514 temp_unit_addr = inst_ptr->unit_addr[inst_ptr->cur_unit_addr_index][0],
515 for (unitno = 0, unitno < MAX_UNIT_COUNT; ++unitno)
516     temp_unit_addr[unitno] = unit_addr[unitno],
517
518 /* Restore unit_addr[1] */
519 temp_unit_addr =
520     inst_ptr->unit_addr[inst_ptr->cur_unit_addr_index + 1][0],
521 for (unitno = 0, unitno < MAX_UNIT_COUNT; ++unitno)
522     temp_unit_addr[unitno] = unit_addr2[unitno],
523
524 /* Restore max_addr and prev_max_addr */
525 for (laneso = 0, laneso < MAX_LANE_COUNT; ++laneso) {
526     inst_ptr->lans_addr[laneso].max_addr = max_addr[laneso],
527     inst_ptr->lans_addr[laneso].prev_max_addr = prev_max_addr[laneso],
528 }
529
530
531 last_patterns_spans_2_blocks(instance)
532
533 INSTANCE_INFO *instance,
534
535 /* Return TRUE if the last pattern of the given instance spans over
536  * the last 2 blocks of the pattern sequence, otherwise return FALSE.
537  * Note that we only have the check 1 lane since the pattern grows
538  * proportionately in each lane.
539  */
540
541 DAB_INFO *dab_ptr,
542 u_long block_addr,
543 u_long temp_unit_addr,
544 u_char laneso,
545 u_char unitno,
546 u_char total_unit,
547
548 dab_ptr = dab_list(instance->dab_info_index),
549 laneso = dab_ptr->unit_location[0].lane_no,
550
551 total_unit = dab_ptr->unit_count_per_lane * dab_ptr->lane_count,
552 temp_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0],
553 block_addr = temp_unit_addr[0] + BLOCK_START_MASK,
554
555 for (unitno = 1, unitno < total_unit; ++unitno) {
556     if (dab_ptr->unit_location[unitno].lane_no == laneso) {
557         if ((temp_unit_addr[unitno] + BLOCK_START_MASK) != block_addr)
558             return(TRUE);
559     }
560 }
561
562 return(FALSE);
563
564
565 copy_instance_info(inst_ptr, fault_ptr)
566
567 INSTANCE_INFO *inst_ptr,
568 INSTANCE_INFO *fault_ptr,
569
570 /* Copy the following information from the instance data structure to the
571  * fault data structure:
572  * PATTERN_COUNT
573  * STATIC_PATTERN_COUNT
574  * COMMON_PATTERN_COUNT
575  * FB_BLOCK_SIZE
576  * FB_BLOCK_ADDR
577  * PTON_LOADED
578  * ETCODES_LOADED
579  * ITCODES_LOADED
580  * SIM_PIN_VALUE
581  * LAST_SAMPLE_VALUE
582  */
583
584 DAB_INFO *dab_ptr,
585 u_char laneso,
586 u_char unitno,
587 u_char total_unit,
588
589 dab_ptr = dab_list(inst_ptr->dab_info_index),
590
591 total_unit = dab_ptr->unit_count,
592
593 fault_ptr->patterns_count = inst_ptr->patterns_count,
594 fault_ptr->static_patterns_count = inst_ptr->static_patterns_count,
595 fault_ptr->common_patterns_count = inst_ptr->patterns_count,
596
597
598 for (laneso = 0, laneso < MAX_LANE_COUNT; ++laneso) {
599     fault_ptr->fb_block_size[laneso] = inst_ptr->fb_block_size[laneso],
600     fault_ptr->fb_block_addr[laneso] = inst_ptr->fb_block_addr[laneso],

```

Copyright 1989  
Logic Modeling SystemsSOURCE PROGRAM  
lm1000/fault.cDATE 5/23/89  
TIME 6:14:37 pmPAGE #  
6/17

LINE #	SOURCE TEXT
601	for (unitno = 0; unitno < total_unit; ++unitno) {
602	fault_ptr->ptrn_loaded[unitno] = inst_ptr->ptrn_loaded[unitno];
603	fault_ptr->hndemb_loaded[unitno] = inst_ptr->ptrn_loaded[unitno];
604	fault_ptr->lcycmdb_loaded[unitno] = inst_ptr->lcycmdb_loaded[unitno];
605	fault_ptr->lcycmdb_loaded[unitno] = 1;     ptr->lcycmdb_loaded[unitno];
606	fault_ptr->sim_pin_value.data[unitno] =
607	inst_ptr->sim_pin_value.data[unitno];
608	fault_ptr->sim_pin_value.hiz[unitno] =
609	inst_ptr->sim_pin_value.hiz[unitno];
610	fault_ptr->sim_pin_value.unknown[unitno] =
611	inst_ptr->sim_pin_value.unknown[unitno];
612	fault_ptr->sim_pin_value.soft[unitno] =
613	inst_ptr->sim_pin_value.soft[unitno];
614	fault_ptr->last_sample_value.data[unitno] =
615	inst_ptr->last_sample_value.data[unitno];
616	fault_ptr->last_sample_value.hiz[unitno] =
617	inst_ptr->last_sample_value.hiz[unitno];
618	fault_ptr->last_sample_value.unknown[unitno] =
619	inst_ptr->last_sample_value.unknown[unitno];
620	fault_ptr->last_sample_value.soft[unitno] =
621	inst_ptr->last_sample_value.soft[unitno];
622	fault_ptr->last_sample_value.soft[unitno] =
623	inst_ptr->last_sample_value.soft[unitno];
624	fault_ptr->last_sample_value.soft[unitno] =
625	inst_ptr->last_sample_value.soft[unitno];
626	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
1/18

```

LINE # SOURCE TEXT
1  /* SCCS_ID: function.c rev 1.7, 5/9/89 at 19:06:33 */
2  #include "common.h"
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "seeprom.h"
7  #include "laser.h"
8  #include "protans.h"
9  #include "lm_sfi.h"
10 #include "lsmnetwork.h"
11 #include "network.h"
12 #include "mod_err.h"
13 #include "nvram.h"
14
15 #ifdef MODELER
16 #include "vrtx.h"
17 #endif
18
19 extern CONNECTION *table_of_conns[];
20
21 /* MML initialization routine */
22 extern void lm_init_vars();
23
24 #define MAX_TEMP_BUF_SIZE (3*10*1024)
25 #define TICKS_PER_SECOND 200
26
27 extern char *lmi_version;
28
29 extern long calculate_pal_count();
30
31 extern u_long total_malloc_size;
32 extern u_long available_malloc_size;
33
34 static u_char modeler_is_locked = FALSE;
35 static char user_with_lock[256];
36
37 void process_begin_session_cmd(user)
38 USER_INFO *user;
39 {
40     char *temp_ptr;
41     char error_string[512];
42     u_short temp;
43     u_short i;
44     char c;
45     version_type sim_version;
46     version_type cnc_version;
47     version_type last_compatible_version;
48
49     DPRINTF(("inside process_begin_session_cmd\n"));
50
51     resetobuf();
52     lm_put_int(BEGIN_SESSION_ANS);
53
54     /* Version number is encoded in simulator_type */
55     sim_version = lm_get_int();
56     cnc_version = SOFTWARE_REVISION_NUMBER;
57     last_compatible_version = LAST_COMPATIBLE_CNC_VERSION;
58
59     switch (sim_version.field.sim_type) {
60     case (LM_FAULT_SIMULATOR & SIM_TYPE_MASK):
61         user->is_fault_simulator = TRUE;
62         break;
63     case (LM_LOGIC_SIMULATOR & SIM_TYPE_MASK):
64         user->is_fault_simulator = FALSE;
65         break;
66     default:
67         lm_queue_message(ERROR_MSG, "internal simulator error: illegal simulator type");
68         end_queue_message();
69         end_put(user->id);
70         return;
71     }
72
73     /* Check Version Number */
74     if ( (sim_version.field.major > cnc_version.field.major)
75         || (sim_version.field.major < last_compatible_version.field.major)
76         || (sim_version.field.major == last_compatible_version.field.major)
77         && (sim_version.field.minor < last_compatible_version.field.minor)) {
78         if (sim_version.field.major > cnc_version.field.major) {
79             lm_queue_message(ERROR_MSG, "Modeler is booted with Core Modeler Code that is incompatible with host application, reboot Modeler with most recent Core Modeler Code release");
80         }
81         else {
82             lm_queue_message(ERROR_MSG, "Modeler is booted with Core Modeler Code that is incompatible with host application, reboot Modeler with previous Core Modeler Code release");
83         }
84         end_queue_message();
85         end_put(user->id);
86         return;
87     }
88
89     temp_ptr = (char *)LM_GET_ADDR(lm_global.conn_ptr);
90     if (strlen(temp_ptr) > 1 & MAX_STRING_LENGTH) {
91         lm_queue_message(ERROR_MSG, "internal simulator error: hostname too long");
92         end_queue_message();
93         end_put(user->id);
94         return;
95     }
96     for (i = 0; (c = lm_get_char()) != '\0'; ++i)
97         user->hostname[i] = c;
98     user->hostname[i] = '\0';
99
100     temp_ptr = (char *)LM_GET_ADDR(lm_global.conn_ptr);
101     if (strlen(temp_ptr) > 1 & MAX_STRING_LENGTH) {
102         lm_queue_message(ERROR_MSG, "internal simulator error: username too long");
103         end_queue_message();
104         end_put(user->id);
105         return;
106     }
107     for (i = 0; (c = lm_get_char()) != '\0'; ++i)
108         user->username[i] = c;
109     user->username[i] = '\0';
110
111     if (modeler_is_locked == TRUE) {
112         DPRINTF(("intruder: %s@%s\n", user->username, user->hostname));
113         if (strlen(user->username, user_with_lock) != 0) {
114             lm_queue_message(ERROR_MSG, "modeler is currently locked by: %s",
115                             user_with_lock);
116             abort_user(user);
117             end_queue_message();
118             end_put(user->id);
119         }
120     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

2/19

```

119
120     if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
121         while (1) {
122             if (lm_queue_message(itmp, error_string) != FAILURE)
123                 break;
124             else
125                 break;
126         }
127     }
128     return;
129 }
130
131 end_queue_message();
132 end_put(user->id);
133 return;
134 }
135
136 void process_create_def_cmd(user)
137 USER_INFO *user;
138 {
139     DEVICE_SPEC *def_ptr;
140     PIN_SPEC *pin_ptr;
141     DAB_INFO *dab_ptr;
142     char *buffer_ptr;
143     u_long pin_count_mark;
144     u_long number;
145     long units;
146     u_short pin_count;
147     u_short actual_pin_count;
148     u_short pinno;
149     u_short def_id_table_index;
150     u_short mtyp_count;
151     u_short errors;
152     u_short warnings;
153     char dab_info_index;
154     int i;
155     char c;
156     DEVICE_SPEC *backend_pass;
157
158     DPRINTF(("inside process_create_def_cmd\n"));
159
160     reset_obuf();
161     lm_put_int(CREATE_DEF_AMS);
162
163     number = lm_get_int();
164     units = lm_get_int();
165
166     if (set_time_scale(user, number, units) == FAILURE) {
167         end_queue_message();
168         end_put(user->id);
169         return;
170     }
171
172     lm_init_vars("HMBL");
173     buffer_ptr = LM_GET_ADDR(lm_global_conn_ptr);
174
175     def_ptr = backend_pass(buffer_ptr);
176     if (def_ptr == NULL) {
177         end_queue_message();
178         end_put(user->id);
179         return;
180     }
181
182     /* Convert all devices to PUBLIC if it is fault simulator. */
183     if (user->is_fault_simulator == TRUE) {
184         def_ptr->device_type = PUBLIC;
185     }
186
187     adjust_delay(user, def_ptr);
188
189     lm_message_types(&errors, &warnings);
190
191     if (errors) {
192         end_queue_message();
193         end_put(user->id);
194         return;
195     }
196
197     if (new_and_link_definition(user, def_ptr,
198                                def_id_table_index) == FAILURE) {
199         free_device(def_ptr);
200         lm_queue_message(ERROR_MSG, "out of memory on modeler for definition");
201         end_queue_message();
202         end_put(user->id);
203         return;
204     }
205
206     if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,
207                                           (u_char)def_ptr->device_type)) != -1) {
208         if ((long)def_ptr->pin_cnt - 1 >=
209             (long)dab_list[dab_info_index]->unit_count * MAX_PIN_PER_UNIT) {
210             rx_definition(user, def_id_table_index);
211             lm_queue_message(ERROR_MSG, "pin number too large; device is only has %d pins",
212                             def_ptr->device_name,
213                             dab_list[dab_info_index]->unit_count * MAX_PIN_PER_UNIT);
214             end_queue_message();
215             end_put(user->id);
216             return;
217         }
218     }
219
220     if (fill_in_extra_data(def_ptr, dab_info_index) == FAILURE) {
221         rx_definition(user, def_id_table_index);
222         lm_queue_message(ERROR_MSG, "out of memory on modeler for definition");
223         end_queue_message();
224         end_put(user->id);
225         return;
226     }
227
228     else {
229         rx_definition(user, def_id_table_index);
230         lm_queue_message(ERROR_MSG, "device name is not found or being used as PRIVATE",
231                         def_ptr->device_name);
232         end_queue_message();
233         end_put(user->id);
234         return;
235     }
236 }
237
238 }

```



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE

5/23/89

PAGE #

TIME

6:14:38 pm

3/20

```

LINE # SOURCE TEXT
239 dab_ptr = dab_list[dab_info_index];
240
241 if (place_edges(def_ptr, dab_ptr) == FAILURE) {
242     ris_definition(user, def_id_table_index);
243     end_queue_message();
244     end_put(user->id);
245     return;
246 }
247
248 if (calc_dab_voltage(def_ptr, dab_list[dab_info_index]) == FAILURE) {
249     ris_definition(user, def_id_table_index);
250     end_queue_message();
251     end_put(user->id);
252     return;
253 }
254
255 if (verify_soft_drive_current(def_ptr,
256     dab_list[dab_info_index]) == FAILURE) {
257     ris_definition(user, def_id_table_index);
258     end_queue_message();
259     end_put(user->id);
260     return;
261 }
262
263 end_queue_message();
264
265 /* write the Definition ID */
266 lm_put_short(def_id_table_index);
267
268 lm_put_char(def_ptr->use_default);
269 lm_put_char(def_ptr->report_miss);
270
271 lm_put_int(def_ptr->default_delay.minimum);
272 lm_put_int(def_ptr->default_delay.typical);
273 lm_put_int(def_ptr->default_delay.maximum);
274
275 /* Return the delay table */
276 mtyz_count = def_ptr->mtyz_cat;
277 lm_put_int(mtyz_count);
278 for (i = 0; i < mtyz_count; ++i) {
279     lm_put_int(def_ptr->mtyz_table[i].minimum);
280     lm_put_int(def_ptr->mtyz_table[i].typical);
281     lm_put_int(def_ptr->mtyz_table[i].maximum);
282 }
283
284 /* write pin count */
285 pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
286 lm_put_int(0);
287
288 /* write pin id's */
289 actual_pin_count = 0;
290 pin_count = def_ptr->pin_cat;
291 pin_ptr = def_ptr->pin_table;
292 for (pinno = 0; pinno < pin_count; ++pinno) {
293     if (pin_ptr->direction == NONE) {
294         ++pin_ptr;
295         continue;
296     }
297
298 #ifdef DEBUG
299     DPRINTF((" Pin name: %15s  PN: %3d  ",
300         pin_ptr->pin_name, pinno));
301
302     switch (pin_ptr->pin_class) {
303     case DATA:
304         DPRINTF(("D "));
305         break;
306     case EVAL:
307         DPRINTF(("E "));
308         break;
309     case STORE:
310         switch (pin_ptr->clk_format) {
311         case DWZ:
312             DPRINTF(("S "));
313             break;
314         case R1:
315             DPRINTF(("R "));
316             break;
317         case R0:
318             DPRINTF(("F "));
319             break;
320         default:
321             DPRINTF(("S? "));
322             break;
323         }
324         break;
325     default:
326         DPRINTF(("? "));
327         break;
328     }
329
330     switch (pin_ptr->direction) {
331     case IN:
332         DPRINTF(("I  \n"));
333         break;
334     case OUT:
335         DPRINTF(("O  \n"));
336         break;
337     case IO:
338         DPRINTF(("IO \n"));
339         break;
340     case POWER:
341         DPRINTF(("PWR \n"));
342         break;
343     case GROUND:
344         DPRINTF(("GND \n"));
345         break;
346     case NC:
347         DPRINTF(("NC  \n"));
348         break;
349     default:
350         DPRINTF(("??  \n"));
351         break;
352     }
353 }
354 #endif
355
356 ++actual_pin_count;
357
358 if ((pin_ptr->direction == POWER) ||

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89 TIME 6:14:38 pm	PAGE # 4/21
LINE #	SOURCE TEXT			
359	(pin_ptr->direction == GROUND)			
360	(pin_ptr->direction == NC)) {			
361	lm_put_short(pname);			
362	lm_put_short(DATA);			
363	lm_put_short(pin_ptr->direction);			
364	for (i = 0; c = pin_ptr->pin_name[i]; ++i)			
365	lm_put_char(c);			
366	lm_put_char('\0');			
367	for (i = 0; c = pin_ptr->pin_number[i]; ++i)			
368	lm_put_char(c);			
369	lm_put_char('\0');			
370	if (pin_ptr->pin_alias == NULL) {			
371	lm_put_char('\0');			
372	}			
373	else {			
374	for (i = 0; c = pin_ptr->pin_alias[i]; ++i)			
375	lm_put_char(c);			
376	lm_put_char('\0');			
377	}			
378	++pin_ptr;			
379	continue;			
380	}			
381	lm_put_short(pname);			
382	switch (pin_ptr->pin_class) {			
383	case DATA:			
384	lm_put_short(pin_ptr->pin_class);			
385	break;			
386	case STORE:			
387	/* Figure out whether it's store both, store rise, or store fall */			
388	switch (pin_ptr->clk_format) {			
389	case DWRZ:			
390	lm_put_short(pin_ptr->pin_class);			
391	break;			
392	case R1:			
393	lm_put_short(EDGE_RISE);			
394	break;			
395	case F0:			
396	lm_put_short(EDGE_FALL);			
397	break;			
398	default:			
399	break;			
400	}			
401	break;			
402	default:			
403	break;			
404	}			
405	lm_put_short(pin_ptr->direction);			
406	for (i = 0; c = pin_ptr->pin_name[i]; ++i)			
407	lm_put_char(c);			
408	lm_put_char('\0');			
409	for (i = 0; c = pin_ptr->pin_number[i]; ++i)			
410	lm_put_char(c);			
411	lm_put_char('\0');			
412	if (pin_ptr->pin_alias == NULL) {			
413	lm_put_char('\0');			
414	}			
415	else {			
416	for (i = 0; c = pin_ptr->pin_alias[i]; ++i)			
417	lm_put_char(c);			
418	lm_put_char('\0');			
419	}			
420	++pin_ptr;			
421	}			
422	LM_PUT_LONG_AT_MARK(pin_count_mark, lm_global_cons_ptr, actual_pin_count);			
423	end_put(user->id);			
424	/* ARGSUSED */			
425	void process_check_dabdef_cmd(user)			
426	USER_INFO *user;			
427	{			
428	DEVICE_SPEC *backend_pass();			
429	DEVICE_SPEC *def_ptr;			
430	char *buffer_ptr;			
431	DPRINTF(("inside process_check_dabdef_cmd\n"));			
432	resetobuf();			
433	lm_put_int(CHECK_DABDEF_ANS);			
434	lm_init_vars("EMBL");			
435	buffer_ptr = LM_GET_ADDR(lm_global_cons_ptr);			
436	def_ptr = backend_pass(buffer_ptr);			
437	if (def_ptr != NULL)			
438	free_device(def_ptr);			
439	end_queue_message();			
440	end_put(user->id);			
441	}			
442	void process_create_instance_cmd(user)			
443	USER_INFO *user;			
444	{			
445	EXTRA_DEVICE_SPEC *extra_def_ptr;			
446	DEVICE_SPEC *def_ptr;			
447	DAB_INFO *dab_ptr;			
448	INSTANCE_INFO *instance;			
449	long def_id;			
450	char *name_ptr;			
451	u_short error_count;			
452	u_short warning_count;			
453	u_short inst_id_table_index;			
454	char dab_info_index;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	5/22

```

LINE #      SOURCE TEXT
479      DPRINTF(("inside process_create_instance_cmd\n"));
480
481      reset_obuf();
482      lm_put_inst(CREATE_INSTANCE_ANS);
483
484      def_id = lm_get_short();
485
486      name_ptr = LM_GET_ADDR(lm_global_conn_ptr);
487
488      /* verify the def_id */
489      if ((def_id < 0) || (def_id >= user->def_table_size)) {
490          lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: id specified",
491                          def_id);
492          end_queue_message();
493          end_put(user->id);
494          return;
495      }
496
497      def_ptr = user->definition[def_id];
498      if (BOGUS_DEFINITION(user, def_ptr)) {
499          lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: id specified",
500                          def_id);
501          end_queue_message();
502          end_put(user->id);
503          return;
504      }
505
506      extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
507      if (extra_def_ptr->dab_ok == FALSE) {
508          lm_queue_message(ERROR_MSG, "Device Adapter was removed");
509          end_queue_message();
510          end_put(user->id);
511          return;
512      }
513
514      if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,
515                                           (u_char)def_ptr->device_type)) == -1) {
516          /* If def_ptr->device_type == PUBLIC, then this error can happen if
517             * someone unplugged the DAB after the definition is created.
518             * If def_ptr->device_type == PRIVATE, then this error can happen if:
519             * - someone else has grabbed the device after the definition is
520               created
521             * - this user has more instances than DAB's.
522             */
523          lm_queue_message(ERROR_MSG, "cannot find device_name: %s",
524                          def_ptr->device_name);
525          end_queue_message();
526          end_put(user->id);
527          return;
528      }
529
530      dab_ptr = dab_list[dab_info_index];
531
532      if (def_ptr->device_type == PRIVATE) {
533          dab_ptr->used_as_private = TRUE;
534      }
535
536      if (new_and_link_instance(user, def_ptr, name_ptr,
537                              dab_ptr->unit_count,
538                              inst_id_table_index,
539                              dab_info_index) == FAILURE) {
540          lm_queue_message(ERROR_MSG, "out of memory on modeler for instance");
541          end_queue_message();
542          end_put(user->id);
543          return;
544      }
545
546      instance = user->instance[inst_id_table_index];
547
548      instance->is_fault = FALSE;
549
550      if (def_ptr->device_type == PRIVATE) {
551          set_private_mode(dab_ptr, TRUE);
552      }
553
554      build_static_patterns_seq(instance);
555
556      lm_message_types(&error_count, &warning_count);
557
558      if (error_count != 0) {
559          /* If there are any errors then release this instance since the host
560             * SFI is not going to create the instance if there are errors.
561             */
562          if (def_ptr->device_type == PRIVATE) {
563              set_private_mode(dab_ptr, FALSE);
564          }
565
566          dab_ptr->used_as_private = FALSE;
567          return_all_ptrs_block(instance);
568          file_instance(user, inst_id_table_index);
569          end_queue_message();
570      }
571
572      else {
573          if (def_ptr->device_type == PRIVATE) {
574              instance->has_history = TRUE;
575              instance->purge_ptrs_on_next_eval = TRUE;
576          }
577
578          dab_ptr->act_inst_count += 1;
579
580          turn_on_in_use(dab_ptr);
581
582          end_queue_message();
583
584          lm_put_short(inst_id_table_index);
585
586          copy_initial_values(instance);
587
588          end_put(user->id);
589      }
590
591      void process_create_fault_cmd(user)
592      USER_INFO user;
593
594      {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89 PAGE #  
TIME 6:14:38 pm 6/23

```

LINE # SOURCE TEXT
599 INSTANCE_INFO *inst_ptr;
600 INSTANCE_INFO *fault_ptr;
601 DAB_INFO *dab_ptr;
602 long inst_id;
603 common_ptrs_count;
604 u_long temp_patterns_count;
605 u_long temp_unit_addr[MAX_UNIT_COUNT];
606 u_long temp_unit_addr2[MAX_UNIT_COUNT];
607 u_long temp_max_addr[MAX_LANE_COUNT];
608 u_long temp_prev_max_addr[MAX_LANE_COUNT];
609 u_short error_count;
610 u_short warning_count;
611 u_short table_index;
612
613 DPRINTF(("inside process_create_fault_cmd\n"));
614
615 reset_obuf();
616 lm_put_int(CREATE_FAULT_MSG);
617
618 inst_id = lm_get_short();
619
620 /* verify the inst_id */
621 if ((inst_id < 0) || (inst_id > user->inst_table_size)) {
622     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
623                     inst_id);
624     end_queue_message();
625     end_put(user->id);
626     return;
627 }
628
629 inst_ptr = user->instance[inst_id];
630 if (BEGUS_INSTANCE(user, inst_ptr)) {
631     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified",
632                     inst_id);
633     end_queue_message();
634     end_put(user->id);
635     return;
636 }
637
638 if (inst_ptr->definition->device_type == PRIVATE) {
639     /* This condition can happen if the user says it's LOGIC_SIMULATOR
640      * and then call lm_create_fault(). If the simulator is FAULT_SIMULATOR
641      * then we are going to convert the device_type to PUBLIC when
642      * we create definitions.
643      */
644     lm_queue_message(ERROR_MSG, "internal simulator error: cannot create fault on a private mode device");
645     end_queue_message();
646     end_put(user->id);
647     return;
648 }
649
650 if (((EXTRA_DEVICE_SPEC *)
651     inst_ptr->definition->extra_data->dab_ok == FALSE)) {
652     lm_queue_message(ERROR_MSG, "Device Adapter was removed");
653     end_queue_message();
654     end_put(user->id);
655     return;
656 }
657
658 dab_ptr = dab_list[inst_ptr->dab_info_index];
659
660 common_ptrs_count = lm_get_int();
661
662 if (common_ptrs_count == -1) {
663     /* The fault patterns should branch out from whatever patterns the
664      * instance has currently.
665      */
666     (void)setup_fault_patterns(user, inst_ptr, table_index);
667 }
668 else {
669     /* Make setup_fault_patterns() believe that the instance only has
670      * "common_ptrs_count" number of patterns. Save the actual
671      * information about the instance patterns in temp_*.
672      */
673     modify_instance_patterns_addr(inst_ptr, common_ptrs_count,
674                                   temp_patterns_count,
675                                   temp_unit_addr,
676                                   temp_unit_addr2,
677                                   temp_max_addr,
678                                   temp_prev_max_addr);
679
680     (void)setup_fault_patterns(user, inst_ptr, table_index);
681
682     restore_instance_patterns_addr(inst_ptr,
683                                   temp_patterns_count,
684                                   temp_unit_addr,
685                                   temp_unit_addr2,
686                                   temp_max_addr,
687                                   temp_prev_max_addr);
688 }
689
690 lm_message_types(error_count, warning_count);
691
692 if (error_count != 0) {
693     /* If there are any errors then release this instance since the host
694      * SRI is not going to create the instance if there are errors.
695      */
696     fault_ptr = user->instance[table_index];
697     return_all_ptrs_b(fault_ptr);
698     release_instance(user, table_index);
699     end_queue_message();
700 }
701 else {
702     ++inst_ptr->fault_count;
703
704     dab_ptr->act_var_count += 1;
705
706     turn_on_in_use(dab_ptr);
707
708     end_queue_message();
709     lm_put_short(table_index);
710 }
711
712 end_put(user->id);
713
714
715 void process_release_def_cmd(user)
716 USER_INFO *user;
717 {
718     DEVICE_SPEC *def_ptr;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
7/24

```

LINE # SOURCE TEXT
719 INSTANCE_INFO *instance;
720 long def_id;
721 u_short i;
722
723 DPRINTF(("inside process_release_def_cmd\n"));
724
725 reset_obuf();
726 lm_put_int(RELEASE_DEF_ANS);
727
728 def_id = lm_get_short();
729
730 /* verify the def_id */
731 if ((def_id < 0) || (def_id >= user->def_table_size)) {
732     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
733                     def_id);
734     end_queue_message();
735     end_put(user->id);
736     return;
737 }
738
739 def_ptr = user->definition[def_id];
740 if (BOGUS_DEFINITION(user, def_ptr)) {
741     lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
742                     def_id);
743     end_queue_message();
744     end_put(user->id);
745     return;
746 }
747
748 /* Check if this definition still has instances/faults */
749 for (i = 0; i < user->inst_table_size; ++i) {
750     instance = user->instance[i];
751
752     if (BOGUS_INSTANCE(user, instance))
753         continue;
754
755     if (instance->definition == def_ptr) {
756         lm_queue_message(ERROR_MSG, "internal simulator error: Definition id: %d still has instances/faults associated with it",
757                         def_id);
758         end_queue_message();
759         end_put(user->id);
760         return;
761     }
762 }
763
764 rls_definition(user, (u_short)def_id);
765
766 end_queue_message();
767 end_put(user->id);
768
769 void process_release_instance_cmd(user)
770 USER_INFO *user;
771 {
772     INSTANCE_INFO *instance;
773     DAB_INFO *dab_ptr;
774     long inst_id;
775
776     DPRINTF(("inside process_release_instance_cmd\n"));
777
778     reset_obuf();
779     lm_put_int(RELEASE_INSTANCE_ANS);
780
781     inst_id = lm_get_short();
782
783     /* verify the inst_id */
784     if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
785         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
786                         inst_id);
787         end_queue_message();
788         end_put(user->id);
789         return;
790     }
791
792     instance = user->instance[inst_id];
793     if (BOGUS_INSTANCE(user, instance)) {
794         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
795                         inst_id);
796         end_queue_message();
797         end_put(user->id);
798         return;
799     }
800
801     if (instance->is_fault) {
802         lm_queue_message(ERROR_MSG, "internal simulator error: instance id: %d is a fault",
803                         inst_id);
804         end_queue_message();
805         end_put(user->id);
806         return;
807     }
808
809     /* Check if this instance still has faults */
810     /* ??? Can't do this with the current structure.
811     * Just make sure this condition is checked on the host.
812     */
813
814     return_all_ptrs_block(instance);
815
816     dab_ptr = dab_list[instance->dab_info_index];
817
818     rls_instance(user, (u_short)inst_id);
819
820     --dab_ptr->act_inst_count;
821
822     if (dab_ptr->used_as_private == TRUE) {
823         dab_ptr->used_as_private = FALSE;
824         set_private_mode(dab_ptr, FALSE);
825     }
826
827     if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
828         turn_off_in_use(dab_ptr);
829
830     end_queue_message();
831     end_put(user->id);
832 }
833
834 void process_release_fault_cmd(user)

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
8/25

```

LINE # SOURCE TEXT
839 USER_INFO *user;
840 {
841     INSTANCE_INFO *fault;
842     INSTANCE_INFO *instance;
843     DAB_INFO *dab_ptr;
844     long fault_id;
845     u_short inst_id;
846     u_short i;
847
848     DPRINTF(("inside process_release_fault_cmd\n"));
849
850     reset_obuf();
851     lm_put_inst(RELEASE_FAULT_ANS);
852
853     fault_id = lm_get_short();
854     inst_id = lm_get_short();
855
856     /* verify the fault_id */
857     if ((fault_id < 0) || (fault_id >= user->inst_table_size)) {
858         lm_queue_message(ERROR_MSG, "internal simulator error: invalid fault id: %d specified",
859             fault_id);
860         end_queue_message();
861         end_put(user->id);
862         return;
863     }
864
865     /* verify the inst_id */
866     if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
867         lm_queue_message(ERROR_MSG, "internal error: invalid inst id: %d specified",
868             inst_id);
869         end_queue_message();
870         end_put(user->id);
871         return;
872     }
873
874     fault = user->instance[fault_id];
875     if (BOGUS_INSTANCE(user, fault)) {
876         lm_queue_message(ERROR_MSG, "internal simulator error: invalid fault id: %d specified",
877             fault_id);
878         end_queue_message();
879         end_put(user->id);
880         return;
881     }
882
883     if (! fault->is_fault) {
884         lm_queue_message(ERROR_MSG, "internal simulator error: fault id: %d is an instance",
885             fault_id);
886         end_queue_message();
887         end_put(user->id);
888         return;
889     }
890
891     instance = user->instance[inst_id];
892     if (BOGUS_INSTANCE(user, instance)) {
893         lm_queue_message(ERROR_MSG, "internal error: invalid inst id: %d specified",
894             inst_id);
895         end_queue_message();
896         end_put(user->id);
897         return;
898     }
899
900     if (instance->is_fault) {
901         lm_queue_message(ERROR_MSG, "internal error: instance id: %d is a fault",
902             inst_id);
903         end_queue_message();
904         end_put(user->id);
905         return;
906     }
907
908     return_all_ptr_block(fault);
909
910     dab_ptr = dab_list(fault->dab_info_index);
911
912     --instance->fault_count;
913
914     rls_instance(user, (u_short) fault_id);
915
916     --dab_ptr->act_var_count;
917
918     if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
919         turn_off_is_use(dab_ptr);
920
921     end_queue_message();
922     end_put(user->id);
923 }
924
925 void process_eval_cmd(user)
926 USER_INFO *user;
927 {
928     INSTANCE_INFO *instance;
929     DEVICE_SPEC *def_ptr;
930     u_short inst_id;
931     u_short error_count;
932     u_short warning_count;
933     u_char replay_count;
934     u_char changed_dac;
935
936     DPRINTF(("inside process_eval_cmd\n"));
937
938     reset_obuf();
939     lm_put_inst(EVAL_ANS);
940
941     inst_id = lm_get_short();
942
943     /* verify inst_id */
944     if (inst_id >= user->inst_table_size) {
945         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
946             inst_id);
947         end_queue_message();
948         end_put(user->id);
949         return;
950     }
951
952     instance = user->instance[inst_id];
953     if (BOGUS_INSTANCE(user, instance)) {
954         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
955             inst_id);
956         end_queue_message();
957         end_put(user->id);
958         return;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
9/26

```

LINE #          SOURCE TEXT
959      }
960
961      def_ptr = instance->definition;
962      if (def_ptr == NULL) {
963          lm_queue_message(ERROR_MSG, "internal error: no definition for instance");
964          end_queue_message();
965          end_put(user->id);
966          return;
967      }
968
969      if (((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok == FALSE) {
970          lm_queue_message(ERROR_MSG, "Device Adapter was Removed");
971          end_queue_message();
972          end_put(user->id);
973          return;
974      }
975
976      if (instance->failed_to_alloc_ptrn == TRUE) {
977          lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory, cannot continue simulation");
978          end_queue_message();
979          end_put(user->id);
980          return;
981      }
982
983      if (instance->fatal_error == TRUE) {
984          lm_queue_message(ERROR_MSG, "fatal error encountered on this instance, cannot continue simulation");
985          end_queue_message();
986          end_put(user->id);
987          return;
988      }
989
990      if (program_dac(def_ptr, (char)instance->dab_info_index,
991          &changed_dac) == FAILURE) {
992          end_queue_message();
993          end_put(user->id);
994          return;
995      }
996
997      #ifndef DRAKE
998      if (start_tmw(def_ptr) == FAILURE) {
999          end_queue_message();
1000          end_put(user->id);
1001          return;
1002      }
1003      #endif
1004
1005      clear_ptrn_bits((char *)gbl_ident_inconsistent_pins,
1006          dab_list(instance->dab_info_index->unit_count);
1007
1008      clear_ptrn_bits((char *)gbl_ident_change,
1009          dab_list(instance->dab_info_index->unit_count);
1010
1011      if (instance->evaluation_count > 0) {
1012          replay_count = instance->sample_count;
1013          --instance->evaluation_count;
1014      }
1015      else
1016          replay_count = 0;
1017
1018      if (copy_in_pin_changes(instance) == FAILURE) {
1019          end_queue_message();
1020          end_put(user->id);
1021          return;
1022      }
1023
1024      run_input_bcode(instance);
1025
1026      #ifdef DEBUC
1027      print_pin_changes(instance);
1028      #endif
1029
1030      if (def_ptr->device_type == PUBLIC) {
1031          if (instance->enable_timing_meas == TRUE) {
1032              (void)tm_evaluate_1_bit_per_pin(user, def_ptr, instance,
1033                  gbl_ident_inconsistent_pins,
1034                  &gbl_steady_state_result,
1035                  &gbl_tmw_steady_state_result,
1036                  gbl_ident_change,
1037                  &changed_dac);
1038          }
1039          else {
1040              (void)evaluate_1_bit_per_pin(def_ptr, instance,
1041                  gbl_ident_inconsistent_pins,
1042                  &gbl_tmw_steady_state_result,
1043                  gbl_ident_change,
1044                  replay_count,
1045                  &changed_dac);
1046          }
1047      }
1048      else {
1049          /* PRIVATE part */
1050          if (instance->had_aborted_pin == TRUE) {
1051              report_aborted_pin(instance);
1052              instance->fatal_error = TRUE;
1053              end_queue_message();
1054              end_put(user->id);
1055              return;
1056          }
1057
1058          if (instance->has_history == FALSE) {
1059              (void)evaluate_private(def_ptr, instance, gbl_ident_change,
1060                  gbl_ident_inconsistent_pins,
1061                  &changed_dac);
1062          }
1063          else {
1064              if (instance->purge_ptrn_on_next_eval == TRUE) {
1065                  if (purge_ptrn(instance) == SUCCESS) {
1066                      (void)evaluate_private(def_ptr, instance, gbl_ident_change,
1067                          gbl_ident_inconsistent_pins,
1068                          &changed_dac);
1069                  }
1070              }
1071              else {
1072                  if (instance->enable_timing_meas == TRUE) {
1073                      (void)tm_evaluate_1_bit_per_pin(user, def_ptr, instance,
1074                          gbl_ident_inconsistent_pins,
1075                          &gbl_steady_state_result,
1076                          &gbl_tmw_steady_state_result,
1077                          gbl_ident_change,
1078                          &changed_dac);

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	10/27

```

1079
1080     else {
1081         (void)evaluate_1_bit_per_pin(def_ptr, instance,
1082                                     gbl_ident_inconsistent_pin,
1083                                     gbl_temp_steady_state_result,
1084                                     gbl_ident_change,
1085                                     replay_count,
1086                                     changed_dac);
1087     }
1088 }
1089
1090 }
1091
1092 lm_message_types(error_count, warning_count);
1093
1094 if (error_count != 0) {
1095     end_queue_message();
1096     end_put(user->id);
1097     return;
1098 }
1099
1100 run_output_bcode(instance);
1101
1102 end_queue_message();
1103
1104 copy_out_pin_changes(instance);
1105
1106 end_put(user->id);
1107 }
1108
1109 void process_save_def_cmd(user)
1110 USER_INFO *user;
1111 {
1112     DEVICE_SPEC *def_ptr;
1113     INSTANCE_INFO *instance;
1114     char *def_buf;
1115     u_long temp;
1116     u_short def_id;
1117     long i;
1118     #ifdef DEBDC
1119     char *temp_ptr;
1120     #endif
1121
1122     extern char *extract_device();
1123
1124     DPRINTF(("inside process_save_def_cmd\n"));
1125
1126     reset_obuf();
1127     lm_put_inst(SAVE_DEF_ANS);
1128
1129     def_id = lm_get_short();
1130
1131     /* verify def_id */
1132     if (def_id >= user->def_table_size) {
1133         lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
1134                         def_id);
1135         end_queue_message();
1136         end_put(user->id);
1137         return;
1138     }
1139
1140     def_ptr = user->definition[def_id];
1141     if (BOCUS_DEFINITION(user, def_ptr)) {
1142         lm_queue_message(ERROR_MSG, "internal simulator error: invalid definition id: %d specified",
1143                         def_id);
1144         end_queue_message();
1145         end_put(user->id);
1146         return;
1147     }
1148
1149     if (user->save_buffer != NULL) {
1150         lm_queue_message(ERROR_MSG, "internal error: left over text from previous save");
1151         end_queue_message();
1152         end_put(user->id);
1153         return;
1154     }
1155
1156     unadjust_delay(user, def_ptr);
1157
1158     def_buf = extract_device(def_ptr);
1159     if (def_buf == NULL) {
1160         lm_queue_message(ERROR_MSG, "internal error: extract_device failed");
1161         end_queue_message();
1162         end_put(user->id);
1163         return;
1164     }
1165
1166     adjust_delay(user, def_ptr);
1167
1168     DPRINTF(("length of extract_device buffer: %d\n", strlen(def_buf)));
1169
1170     #ifdef DEBDC
1171     #ifdef MODELX
1172     temp_ptr = def_buf;
1173     while (*temp_ptr)
1174         ac_putc(*temp_ptr++);
1175     ac_putc('\n');
1176     #else
1177     temp_ptr = def_buf;
1178     while (*temp_ptr)
1179         DPRINTF((" %c", *temp_ptr++));
1180     DPRINTF((" \n"));
1181     #endif
1182     #endif
1183
1184     end_queue_message();
1185
1186     temp = 0;
1187     for (i = 0; i < user->inst_table_size; ++i) {
1188         instance = user->instance[i];
1189         if (BOCUS_INSTANCE(user, instance))
1190             continue;
1191
1192         if (instance->definition != def_ptr)
1193             continue;
1194
1195         if (instance->is_fault == FALSE)
1196             temp += instance->pattern_count;
1197         else
1198             temp += instance->pattern_count -

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89 PAGE #  
TIME 6:14:38 pm 11/28

```

1199      instance->common_patterns_count;
1200  }
1201
1202  lm_put_int(temp); /* device pattern count */
1203  lm_put_int(strlen(def_buf)); /* total size of the def_buf */
1204
1205  for (i = 0; def_buf[i] != '\0' && i < SAVE_REST_PTRN_BUFFER_LIMIT; ++i) {
1206      lm_put_char(def_buf[i]);
1207  }
1208  lm_put_char('\0');
1209
1210  if (i < SAVE_REST_PTRN_BUFFER_LIMIT) {
1211      /* The whole definition text fits into the network buffer */
1212      user->save_buffer = NULL;
1213      user->save_buffer_offset = -1;
1214      DPRINTF(def_buf);
1215  }
1216  else {
1217      /* There are more to come */
1218      user->save_buffer = def_buf;
1219      user->save_buffer_offset = i;
1220  }
1221
1222  end_put(user->fd);
1223  }
1224
1225  void process_save_def_cont_cmd(user)
1226  USER_INFO *user;
1227  {
1228      char *def_buf;
1229      long i;
1230      long limit;
1231
1232      DPRINTF(("inside process_save_def_cont_cmd\n"));
1233
1234      reset_obuf();
1235      lm_put_int(SAVE_DEF_CONT_ANS);
1236
1237      if (user->save_buffer == NULL) {
1238          lm_queue_message(ERROR_MSG, "internal error: no more definition text to return");
1239          end_queue_message();
1240          end_put(user->fd);
1241          return;
1242      }
1243      end_queue_message();
1244
1245      def_buf = user->save_buffer;
1246
1247      limit = user->save_buffer_offset + SAVE_REST_PTRN_BUFFER_LIMIT;
1248      for (i = user->save_buffer_offset; def_buf[i] != '\0' && i < limit; ++i) {
1249          lm_put_char(def_buf[i]);
1250      }
1251      lm_put_char('\0');
1252
1253      if (i < limit) {
1254          user->save_buffer = NULL;
1255          user->save_buffer_offset = -1;
1256          DPRINTF(def_buf);
1257      }
1258      else {
1259          user->save_buffer_offset = i;
1260      }
1261      end_put(user->fd);
1262  }
1263
1264  void process_save_ptrn_cmd(user)
1265  USER_INFO *user;
1266  {
1267      INSTANCE_INFO *inst_ptr;
1268      DEVICE_SPEC *def_ptr;
1269      PIN_SPEC *pin_def;
1270      PIN_INFO *pin;
1271      DAB_INFO *dab_ptr;
1272      u_long patterns_count;
1273      u_short inst_id;
1274      u_long pin_count_mark;
1275      u_long patterns_count_follows_mark;
1276      u_long out_buffer_remaining_size;
1277      u_short pinno;
1278      u_short pin_count;
1279      u_char device_ptrn_size; /* size of device pattern is bytes */
1280
1281      DPRINTF(("inside process_save_ptrn_cmd\n"));
1282
1283      user->save_state = SENT_RECV_NOTHING;
1284
1285      reset_obuf();
1286      lm_put_int(SAVE_PTRN_ANS);
1287
1288      inst_id = lm_get_short();
1289
1290      /* verify inst_id */
1291      if (inst_id >= user->inst_table_size) {
1292          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1293                          inst_id);
1294          end_queue_message();
1295          end_put(user->fd);
1296          return;
1297      }
1298
1299      inst_ptr = user->instance[inst_id];
1300      if (BOGUS_INSTANCE(user, inst_ptr)) {
1301          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1302                          inst_id);
1303          end_queue_message();
1304          end_put(user->fd);
1305          return;
1306      }
1307
1308      dab_ptr = dab_list[inst_ptr->dab_info_index];
1309
1310      if (inst_ptr->is_fault == FALSE) {
1311          /* This is an instance */
1312
1313          /* The patterns_count includes the last pattern which is a variable
1314           * pattern. This variable pattern is the same as
1315           * INSTANCE_INFO.ptrn_loaded, so we don't have to store this in the
1316           * save set file. Therefore we need to subtract it by
1317           * unit_count_per_line.
1318           */

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
12/29

```

1319 user->pattern_to_send_count = inst_ptr->pattern_count -
1320 inst_ptr->static_pattern_count -
1321 dab_ptr->unit_count_per_lane;
1322
1323 }
1324 else {
1325     /* This is a fault */
1326     /* We don't need to subtract it by unit_count_per_lane since both
1327      * pattern_count and common_pattern_count includes this variable
1328      * pattern already.
1329     user->pattern_to_send_count = inst_ptr->pattern_count -
1330     inst_ptr->common_pattern_count;
1331     }
1332
1333     end_queue_message();
1334
1335     lm_put_char(dab_ptr->unit_count);
1336
1337     /* The following is a total DEVICE ptrn count, and pattern_to_send_count
1338      * is the number of lane patterns. Therefore we need to divide it by
1339      * unit_count_per_lane.
1340     lm_put_int((user->pattern_to_send_count / dab_ptr->unit_count_per_lane), );
1341
1342     /* write common_ptrn_count */
1343     if (inst_ptr->is_fault == FALSE)
1344         lm_put_int(-1);
1345     else {
1346         /* subtract by unit_count_per_lane because common_ptrn_count includes
1347          * the last pattern which is a variable pattern.
1348         lm_put_int((inst_ptr->common_pattern_count -
1349         inst_ptr->static_pattern_count -
1350         dab_ptr->unit_count_per_lane) /
1351         dab_ptr->unit_count_per_lane);
1352     }
1353
1354     lm_put_char(inst_ptr->check_input_count);
1355     lm_put_char(inst_ptr->first_eval);
1356     lm_put_char(inst_ptr->sample_count);
1357     lm_put_int(inst_ptr->evaluation_count);
1358     lm_put_char(inst_ptr->purge_ptrn_on_next_eval);
1359     lm_put_char(inst_ptr->has_history);
1360     lm_put_char(inst_ptr->use_2_bit_per_pin);
1361     lm_put_char(inst_ptr->enable_timing_meas);
1362     lm_put_char(inst_ptr->had_shorted_pin);
1363
1364     pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1365     lm_put_short(0); /* dummy pin_count */
1366
1367     def_ptr = inst_ptr->definition;
1368     pin_count = 0;
1369     for (pinno = 0; pinno < def_ptr->pin_cnt; ++pinno) {
1370         pin_def = &def_ptr->pin_table[pinno];
1371
1372         if ((pin_def->direction == NONE) ||
1373             (pin_def->direction == POWER) ||
1374             (pin_def->direction == GROUND) ||
1375             (pin_def->direction == NC))
1376             continue;
1377
1378         ++pin_count;
1379         pin = &inst_ptr->pin_info_table[pinno];
1380
1381         lm_put_short(pinno);
1382         lm_put_char(pin->old_raw);
1383         lm_put_char(pin->old_filtered);
1384         lm_put_char(pin->new_raw);
1385         lm_put_char(pin->new_filtered);
1386         lm_put_int((pin->pin_time));
1387         lm_put_char(pin->uninitialized_pin);
1388     }
1389
1390     LM_PUT_SHORT_AT_MARK(pin_count_mark, lm_global_conn_ptr, pin_count);
1391
1392     pattern_count_follows_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1393     lm_put_int(0); /* dummy pattern_count_follows */
1394
1395     out_buffer_remaining_size = SAVE_REST_PTRN_BUFFER_LIMIT;
1396
1397     device_ptrn_size = dab_ptr->unit_count * sizeof(PTRN_BITS);
1398
1399     pattern_count = 0;
1400     while (out_buffer_remaining_size > device_ptrn_size) {
1401         if (send_pattern(user, inst_ptr) == FALSE)
1402             break;
1403
1404         ++pattern_count;
1405         out_buffer_remaining_size -= device_ptrn_size;
1406     }
1407
1408     LM_PUT_LONG_AT_MARK(pattern_count_follows_mark, lm_global_conn_ptr,
1409     pattern_count);
1410
1411     DPRINTF(("pattern_count_follows: %d\n", pattern_count));
1412     end_put(user->fd);
1413 }
1414
1415 void process_save_ptrn_count_cmd(user)
1416 USER_INFO *user;
1417 {
1418     INSTANCE_INFO *inst_ptr;
1419     DAB_INFO *dab_ptr;
1420     u_long pattern_count;
1421     u_short inst_id;
1422     u_long pattern_count_follows_mark;
1423     u_long out_buffer_remaining_size;
1424     u_char device_ptrn_size; /* size of device pattern is bytes */
1425
1426     DPRINTF(("inside process_save_ptrn_count_cmd\n"));
1427
1428     reset_obuf();
1429     lm_put_int(SAVE_PTRN_COUNT_ANS);
1430
1431     inst_id = lm_get_short();
1432
1433     /* verify inst_id */
1434     if (inst_id >= user->inst_table_size) {

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE

5/23/89

PAGE #

TIME

6:14:38 pm

13/30

```

1439     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1440                       inst_id);
1441     end_queue_message();
1442     end_put(user->id);
1443     return;
1444 }
1445
1446 inst_ptr = user->instance[inst_id];
1447 if (BOGUS_INSTANCE(user, inst_ptr)) {
1448     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1449                       inst_id);
1450     end_queue_message();
1451     end_put(user->id);
1452     return;
1453 }
1454
1455 dab_ptr = dab_list[inst_ptr->dab_info_index];
1456
1457 end_queue_message();
1458
1459 pattern_count_follows_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1460 lm_put_int(0); /* dummy count */
1461
1462 out_buffer_remaining_size = SAVE_REST_PTRN_BUFFER_LIMIT;
1463
1464 device_ptrn_size = dab_ptr->unit_count * sizeof(PTRN_BITS);
1465
1466 pattern_count = 0;
1467 while (out_buffer_remaining_size > device_ptrn_size) {
1468     if (send_patterns(user, inst_ptr) == FALSE)
1469         break;
1470     ++pattern_count;
1471     out_buffer_remaining_size -= device_ptrn_size;
1472 }
1473
1474 LM_PUT_LONG_AT_MARK(pattern_count_follows_mark, lm_global_conn_ptr,
1475                     pattern_count);
1476
1477 DPRINTF(("pattern_count_follows: %d\n", pattern_count));
1478
1479 end_put(user->id);
1480 }
1481
1482 void process_restore_inst_cmd(user)
1483 USER_INFO *user;
1484 {
1485     INSTANCE_INFO *instance;
1486     PIN_INFO *pin;
1487     u_short inst_id;
1488     u_short pin_count;
1489     u_short pin_number;
1490
1491     DPRINTF(("inside process_restore_inst_cmd\n"));
1492
1493     reset_cbuf();
1494     lm_put_int(RESTORE_INST_ANS);
1495
1496     inst_id = lm_get_short();
1497
1498     /* verify inst_id */
1499     if (inst_id >= user->inst_table_size) {
1500         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1501                           inst_id);
1502         end_queue_message();
1503         end_put(user->id);
1504         return;
1505     }
1506
1507     instance = user->instance[inst_id];
1508     if (BOGUS_INSTANCE(user, instance)) {
1509         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1510                           inst_id);
1511         end_queue_message();
1512         end_put(user->id);
1513         return;
1514     }
1515
1516     instance->check_input_t_count = lm_get_char();
1517     instance->first_eval = lm_get_char();
1518     instance->sample_count = lm_get_char();
1519     instance->evaluation_count = lm_get_int();
1520     instance->purge_ptrn_on_next_eval = lm_get_char();
1521     instance->has_history = lm_get_char();
1522     instance->use_2_bit_per_pin = lm_get_char();
1523     instance->enable_timing_meas = lm_get_char();
1524     instance->had_shorted_pin = lm_get_char();
1525
1526     pin_count = lm_get_short();
1527
1528     for (i = 0; i < pin_count; ++i) {
1529         pin_number = lm_get_short();
1530         pin = instance->pin_info_table[pin_number];
1531         pin->old_raw = lm_get_char();
1532         pin->old_filtered = lm_get_char();
1533         pin->new_raw = lm_get_char();
1534         pin->new_filtered = lm_get_char();
1535         pin->aim_time = lm_get_int();
1536         pin->uninitialized_pin = lm_get_char();
1537     }
1538
1539     end_queue_message();
1540     end_put(user->id);
1541 }
1542
1543 void process_restore_ptrn_cmd(user)
1544 USER_INFO *user;
1545 {
1546     INSTANCE_INFO *instance;
1547     DAB_INFO *dab_ptr;
1548     u_short pattern_count_follows;
1549     u_short inst_id;
1550     u_short unit_count;
1551     i;
1552
1553     DPRINTF(("inside process_restore_ptrn_cmd\n"));
1554 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
14/31

```

1559 reset_obj();
1560 lm_put_int(MESSAGE_PTRN_ANS);
1561
1562 inst_id = lm_get_short();
1563 unit_count = lm_get_int();
1564 pattern_count_follows = lm_get_int();
1565
1566 /* verify inst_id */
1567 if (inst_id >= user->inst_table_size) {
1568     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1569                     inst_id);
1570     end_queue_message();
1571     end_put(user->fd);
1572     return;
1573 }
1574
1575 instance = user->instance[inst_id];
1576 if (BOGUS_INSTANCE(user, instance)) {
1577     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
1578                     inst_id);
1579     end_queue_message();
1580     end_put(user->fd);
1581     return;
1582 }
1583
1584 dab_ptr = dab_list(instance->dab_info_index);
1585
1586 if (dab_ptr->unit_count != unit_count) {
1587     /* ??? error */
1588     lm_queue_message(ERROR_MSG, "internal error: current unit count does not equal saved unit count");
1589     end_queue_message();
1590     end_put(user->fd);
1591     return;
1592 }
1593
1594 if (instance->failed_to_alloc_ptrn == TRUE) {
1595     lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory, cannot restore");
1596     end_queue_message();
1597     end_put(user->fd);
1598     return;
1599 }
1600
1601 if (pattern_count_follows == 0) {
1602     instance->restore_state = SENT_RECV_NOTHING;
1603
1604     /* write the variable pattern */
1605     write_pattern(instance,
1606                 instance->unit_addr[instance->cur_unit_addr_index][0],
1607                 instance->ptrn_loaded);
1608     end_queue_message();
1609     end_put(user->fd);
1610     return;
1611 }
1612
1613 for (i = 0; i < pattern_count_follows; ++i) {
1614     if (restore_inst_pattern(instance, unit_count) == FAILURE)
1615         break;
1616 }
1617
1618 end_queue_message();
1619 end_put(user->fd);
1620
1621 void process_ptrn_hist_cmd(user)
1622 USER_INFO *user;
1623 {
1624     INSTANCE_INFO *instance;
1625     u_short inst_id;
1626     u_char command;
1627
1628     DPRINTF(("inside process_ptrn_hist_cmd\n"));
1629
1630     reset_obj();
1631     lm_put_int(PTRN_HIST_ANS);
1632
1633     inst_id = lm_get_short();
1634
1635     /* verify inst_id */
1636     if (inst_id >= user->inst_table_size) {
1637         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
1638                         inst_id);
1639         end_queue_message();
1640         end_put(user->fd);
1641         return;
1642     }
1643
1644     instance = user->instance[inst_id];
1645     if (BOGUS_INSTANCE(user, instance)) {
1646         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
1647                         inst_id);
1648         end_queue_message();
1649         end_put(user->fd);
1650         return;
1651     }
1652
1653     if (instance->definition->device_type == PUBLIC) {
1654         end_queue_message();
1655         end_put(user->fd);
1656         return;
1657     }
1658
1659     switch (command = lm_get_char()) {
1660     case LM_KEEP_PATTERNS:
1661         if (instance->has_history == TRUE)
1662             instance->purge_ptrn_on_next_eval = FALSE;
1663         else
1664             lm_queue_message(ERROR_MSG, "internal simulator error: instance id: %d does not have history",
1665                             inst_id);
1666         break;
1667     case LM_DONT_KEEP_PATTERNS:
1668         if (instance->has_history == TRUE) {
1669             instance->purge_ptrn_on_next_eval = TRUE;
1670             if (instance->enable_timing_meas == TRUE) {
1671                 lm_queue_message(WARNING_MSG, "timing measurement is turned off on private instance: %s because pattern history were deleted",
1672                                 instance->device_info_string);
1673             }
1674         }
1675     }
1676 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	15/32

```

1679     }
1680     break;
1681     default:
1682         lm_queue_message(ERROR_MSG, "internal simulator error: illegal command: %d",
1683             command);
1684         break;
1685     }
1686     end_queue_message();
1687     end_put(user->id);
1688 }
1689
1690 void process_lm_modeler_cmd(user)
1691 USER_INFO *user;
1692 {
1693     u_long      temp;
1694     u_long      value;
1695     u_short     attrib;
1696     u_char      lane0;
1697     u_char      panno;
1698     u_char      slotno;
1699     u_char      i;
1700
1701     DPRINTF(("inside process_lm_modeler_cmd\n"));
1702
1703     resetobuf();
1704     lm_put_int(INQ_MODELER_ANS);
1705
1706     switch (attrib = lm_get_int()) {
1707     case LM_TOTAL_PATTERNS:
1708         end_queue_message();
1709         temp = 0;
1710         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1711             for (panno = 0; panno < MAX_PAN_COUNT; ++panno)
1712                 if (system_config->lane[lane0]->pan[panno] != NULL)
1713                     temp += system_config->lane[lane0]->pan[panno]->pan_size;
1714         lm_put_int(temp);
1715         break;
1716     case LM_TOTAL_MODELER_MEMORY:
1717         end_queue_message();
1718         lm_put_int(total_malloc_size);
1719         break;
1720     case LM_AVAILABLE_PATTERNS:
1721         end_queue_message();
1722         temp = count_avail_patterns((u_char)MAX_LANE_COUNT) * PTH_PER_BLOCK;
1723         lm_put_int(temp);
1724         break;
1725     case LM_AVAILABLE_MODELER_MEMORY:
1726         end_queue_message();
1727         lm_put_int(available_malloc_size);
1728         break;
1729     case LM_NUMBER_OF_USERS:
1730         end_queue_message();
1731         temp = 0;
1732         for (i = 0; i < MAX_USER_COUNT; ++i)
1733             if (user_info_array[i]->active == TRUE)
1734                 ++temp;
1735         lm_put_int(temp);
1736         break;
1737     case LM_WHICH_USER_AM_I:
1738         end_queue_message();
1739         for (i = 0; i < MAX_USER_COUNT; ++i)
1740             if (user_info_array[i] == user)
1741                 break;
1742         lm_put_int(i);
1743         break;
1744     case LM_NUMBER_OF_LANES:
1745         end_queue_message();
1746         lm_put_int(system_config->phy_lane_count);
1747         break;
1748     case LM_NUMBER_OF_SLOTS_PER_LANE:
1749         end_queue_message();
1750         lm_put_int((long)system_config->slot_count);
1751         break;
1752     case LM_NUMBER_OF_PACS:
1753         end_queue_message();
1754         temp = 0;
1755         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1756             if (system_config->lane[lane0]->pac_present == TRUE)
1757                 ++temp;
1758         lm_put_int(temp);
1759         break;
1760     case LM_NUMBER_OF_PANs:
1761         end_queue_message();
1762         temp = 0;
1763         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1764             for (panno = 0; panno < MAX_PAN_COUNT; ++panno)
1765                 if (system_config->lane[lane0]->pan[panno] != NULL)
1766                     ++temp;
1767         lm_put_int(temp);
1768         break;
1769     case LM_NUMBER_OF_PELs:
1770         end_queue_message();
1771         temp = 0;
1772         for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
1773             for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno)
1774                 if (system_config->lane[lane0]->pel[slotno] != NULL)
1775                     ++temp;
1776         lm_put_int(temp);
1777         break;
1778     case LM_NUMBER_OF_DABs:
1779         end_queue_message();
1780         temp = 0;
1781         for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
1782             if (dab_list[i] != NULL) {
1783                 temp += dab_list[i]->segment_count;
1784             }
1785         lm_put_int(temp);
1786         break;
1787     case LM_NUMBER_OF_PUBLIC_DEVICES:
1788         end_queue_message();
1789         temp = 0;
1790         for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
1791             if (dab_list[i] != NULL) {
1792                 if (dab_list[i]->used_as_private == FALSE) {
1793                     ++temp;
1794                 }
1795             }
1796     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89	PAGE # 16/33
LINE #	SOURCE TEXT			
1799	}			
1800	lm_put_int(temp);			
1801	break;			
1802	case LM_NUMBER_OF_PRIVATE_DEVICES:			
1803	{			
1804	temp = 0;			
1805	for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)			
1806	{			
1807	if (dab_list[i] != NULL) {			
1808	if (dab_list[i] == used_as_private == TRUE) {			
1809	++temp;			
1810	}			
1811	}			
1812	lm_put_int(temp);			
1813	break;			
1814	case LM_NUMBER_OF_ACTIVE_INSTANCES:			
1815	{			
1816	temp = 0;			
1817	for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)			
1818	{			
1819	if (dab_list[i] != NULL)			
1820	temp += dab_list[i] == act_inst_count;			
1821	}			
1822	lm_put_int(temp);			
1823	break;			
1824	case LM_NUMBER_OF_ACTIVE_FAULTS:			
1825	{			
1826	temp = 0;			
1827	for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)			
1828	{			
1829	if (dab_list[i] != NULL)			
1830	temp += dab_list[i] == act_var_count;			
1831	}			
1832	lm_put_int(temp);			
1833	break;			
1834	case LM_SOFTWARE_REVISION_NUMBER:			
1835	{			
1836	lm_put_int(SOFTWARE_REVISION_NUMBER);			
1837	break;			
1838	case LM_PASSWORD_CHECK:			
1839	{			
1840	if (Check_password(user) == TRUE) {			
1841	{			
1842	end_queue_message();			
1843	lm_put_int(LM_TRUE);			
1844	}			
1845	else {			
1846	{			
1847	end_queue_message();			
1848	lm_put_int(LM_FALSE);			
1849	}			
1850	break;			
1851	case LM_GET_VERSION_STRING_ADDR:			
1852	{			
1853	end_queue_message();			
1854	temp = (u_long)line1_version;			
1855	lm_put_int(temp);			
1856	break;			
1857	default:			
1858	{			
1859	if (epoch_info(attrb, avalue) == FAILURE) {			
1860	{			
1861	lm_queue_message(ERROR_MSG, "illegal attribute: %d",			
1862	attrb);			
1863	end_queue_message();			
1864	}			
1865	else {			
1866	{			
1867	end_queue_message();			
1868	lm_put_int(value);			
1869	}			
1870	break;			
1871	}			
1872	end_put(user->fd);			
1873	}			
1874	/* ARGUSED */			
1875	void process_lm_user_list_cmd(user)			
1876	USER_INFO *user;			
1877	{			
1878	u_short temp;			
1879	u_short j;			
1880	u_char i;			
1881	char c;			
1882	DPRINTF(("inside process_lm_user_list_cmd\n"));			
1883	reset_cbuf();			
1884	lm_put_int(LM_USER_LIST_CMD);			
1885	end_queue_message();			
1886	{			
1887	temp = 0;			
1888	for (i = 0; i < MAX_USER_COUNT; ++i)			
1889	{			
1890	if (user_info_array[i] == active == TRUE)			
1891	++temp;			
1892	}			
1893	lm_put_int(temp);			
1894	/* Put the user numbers to the Network buffer */			
1895	for (i = 0; i < MAX_USER_COUNT; ++i) {			
1896	{			
1897	if (user_info_array[i] == active == TRUE) {			
1898	{			
1899	for (j = 0; j < MAX_USER_COUNT; ++j) {			
1900	{			
1901	if (user_info_array[i] == active == TRUE) {			
1902	{			
1903	for (j = 0; j < MAX_USER_COUNT; ++j) {			
1904	{			
1905	if (user_info_array[i] == active == TRUE) {			
1906	{			
1907	for (j = 0; j < MAX_USER_COUNT; ++j) {			
1908	{			
1909	if (user_info_array[i] == active == TRUE) {			
1910	{			
1911	for (j = 0; j < MAX_USER_COUNT; ++j) {			
1912	{			
1913	if (user_info_array[i] == active == TRUE) {			
1914	{			
1915	for (j = 0; j < MAX_USER_COUNT; ++j) {			
1916	{			
1917	if (user_info_array[i] == active == TRUE) {			
1918	{			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

17/34

LINE # SOURCE TEXT

```

1919 USER INFO *userx;
1920 CONNECTION *conn;
1921 long userno;
1922 long attrib;
1923 u_long temp;
1924 u_char i;
1925
1926 DPRINTF(("inside process_lm_user_cmd\n"));
1927
1928 reset_out();
1929 lm_put_int(IMO_USER_AMS);
1930
1931 userno = lm_get_int();
1932 attrib = lm_get_int();
1933
1934 if (userno >= MAX_USER_COUNT) {
1935     lm_queue_message(ERROR_MSG, "invalid user number: td specified",
1936                     userno);
1937     end_queue_message();
1938     end_put(user->id);
1939     return;
1940 }
1941
1942 if (user_info_array[userno]->active == FALSE) {
1943     lm_queue_message(ERROR_MSG, "user number: td does not exist",
1944                     userno);
1945     end_queue_message();
1946     end_put(user->id);
1947     return;
1948 }
1949
1950 userx = user_info_array[userno];
1951
1952 switch (attrib) {
1953 case LM_NUMBER_OF_PATTERNS_ALLOCATED:
1954     end_queue_message();
1955     temp = 0;
1956     for (i = 0; i < userx->inst_table_size; ++i) {
1957         instance = userx->instance[i];
1958         if (BOGUS_INSTANCE(userx, instance))
1959             continue;
1960
1961         if (instance->is_fault == FALSE)
1962             temp += ((instance->pattern_count + PTRN_PER_BLOCK) /
1963                     PTRN_PER_BLOCK) * PTRN_PER_BLOCK;
1964         else
1965             temp += ((instance->pattern_count -
1966                     instance->common_pattern_count + PTRN_PER_BLOCK) /
1967                     PTRN_PER_BLOCK) * PTRN_PER_BLOCK;
1968     }
1969     lm_put_int(temp);
1970     break;
1971 case LM_NUMBER_OF_PATTERNS:
1972     end_queue_message();
1973     temp = 0;
1974     for (i = 0; i < userx->inst_table_size; ++i) {
1975         instance = userx->instance[i];
1976         if (BOGUS_INSTANCE(userx, instance))
1977             continue;
1978
1979         if (instance->is_fault == FALSE)
1980             temp += instance->pattern_count;
1981         else
1982             temp += instance->pattern_count -
1983                     instance->common_pattern_count;
1984     }
1985     lm_put_int(temp);
1986     break;
1987 case LM_NUMBER_OF_PUBLIC_DEVICES:
1988     end_queue_message();
1989     temp = 0;
1990     for (i = 0; i < userx->def_table_size; ++i) {
1991         definition = userx->definition[i];
1992         if (BOGUS_DEFINITION(userx, definition))
1993             continue;
1994
1995         if (definition->device_type == PUBLIC)
1996             ++temp;
1997     }
1998     lm_put_int(temp);
1999     break;
2000 case LM_NUMBER_OF_PRIVATE_DEVICES:
2001     end_queue_message();
2002     temp = 0;
2003     for (i = 0; i < userx->def_table_size; ++i) {
2004         definition = userx->definition[i];
2005
2006         if (BOGUS_DEFINITION(userx, definition))
2007             continue;
2008
2009         if (definition->device_type == PRIVATE)
2010             ++temp;
2011     }
2012     lm_put_int(temp);
2013     break;
2014 case LM_NUMBER_OF_ACTIVE_INSTANCES:
2015     end_queue_message();
2016     temp = 0;
2017     for (i = 0; i < userx->inst_table_size; ++i) {
2018         instance = userx->instance[i];
2019         if (BOGUS_INSTANCE(userx, instance))
2020             continue;
2021
2022         if (instance->is_fault == FALSE)
2023             ++temp;
2024     }
2025     lm_put_int(temp);
2026     break;
2027 case LM_NUMBER_OF_ACTIVE_FAULTS:
2028     end_queue_message();
2029     temp = 0;
2030     for (i = 0; i < userx->inst_table_size; ++i) {
2031         instance = userx->instance[i];
2032
2033         if (BOGUS_INSTANCE(userx, instance))
2034             continue;
2035
2036         if (instance->is_fault == TRUE)
2037             ++temp;
2038     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89	PAGE # 18/35
LINE #		SOURCE TEXT		
2039		lm_put_int(temp);		
2040		break;		
2041		case LM_SECONDS SINCE LAST RESPONSE:		
2042		end_queue_message();		
2043		conn = table_of_conn(userno);		
2044		lm_put_int((lm_tick - conn->time_at_last_response) / TICKS_PER_SECOND);		
2045		break;		
2046		default:		
2047		lm_queue_message(ERROR_MSG, "illegal attribute: %d",		
2048		attrib);		
2049		end_queue_message();		
2050		break;		
2051		}		
2052		end_put(user->id);		
2053		}		
2054				
2055		/* ARGSUSED */		
2056		void process_inq_lane_cmd(user)		
2057		USER_INFO *user;		
2058		{		
2059		long lanesum;		
2060		u_char panno;		
2061		u_char slotno;		
2062		u_long temp;		
2063		u_long attribute;		
2064		DPRINTF(("inside process_inq_lane_cmd\n"));		
2065		reset_obuf();		
2066		lm_put_int(INQ_LANE_ANS);		
2067		lanesum = lm_get_int();		
2068		if ((lanesum < 0)		
2069		(lanesum >= system_config->phy_lane_count)) {		
2070		lm_queue_message(ERROR_MSG, "illegal lane: %d",		
2071		lanesum);		
2072		end_queue_message();		
2073		end_put(user->id);		
2074		return;		
2075		}		
2076				
2077		attribute = lm_get_int();		
2078		if ((attribute != LM_IS_LANE_USABLE) &&		
2079		(system_config->lane[lanesum]->pac_present == FALSE)) {		
2080		lm_queue_message(ERROR_MSG, "lane: %c is not usable",		
2081		'A' + lanesum);		
2082		end_queue_message();		
2083		end_put(user->id);		
2084		return;		
2085		}		
2086				
2087		switch (attribute) {		
2088		case LM_IS_LANE_USABLE:		
2089		end_queue_message();		
2090		if (system_config->lane[lanesum]->pac_present == TRUE)		
2091		lm_put_int(LM_TRUE);		
2092		else		
2093		lm_put_int(LM_FALSE);		
2094		break;		
2095		case LM_TOTAL PATTERNS:		
2096		end_queue_message();		
2097		temp = 0;		
2098		for (panno = 0; panno < MAX_PAM_COUNT; ++panno)		
2099		if (system_config->lane[lanesum]->pan[panno] != NULL)		
2100		temp += system_config->lane[lanesum]->pan[panno]->mem_size;		
2101		lm_put_int(temp);		
2102		break;		
2103		case LM_AVAILABLE PATTERNS:		
2104		end_queue_message();		
2105		temp = count_avail_pattern((u_char)lanesum) * PLEN_PER_BLOCK;		
2106		lm_put_int(temp);		
2107		break;		
2108		case LM_NUMBER OF PAMS:		
2109		end_queue_message();		
2110		temp = 0;		
2111		for (panno = 0; panno < MAX_PAM_COUNT; ++panno)		
2112		if (system_config->lane[lanesum]->pan[panno] != NULL)		
2113		++temp;		
2114		lm_put_int(temp);		
2115		break;		
2116		case LM_NUMBER OF PELS:		
2117		end_queue_message();		
2118		temp = 0;		
2119		for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno)		
2120		if (system_config->lane[lanesum]->pel[slotno] != NULL)		
2121		++temp;		
2122		lm_put_int(temp);		
2123		break;		
2124		case LM_NUMBER OF POPULATED PELS:		
2125		end_queue_message();		
2126		lm_put_int(count_pel_count((u_char)lanesum));		
2127		break;		
2128		default:		
2129		lm_queue_message(ERROR_MSG, "illegal attribute: %d",		
2130		attribute);		
2131		end_queue_message();		
2132		break;		
2133		}		
2134		end_put(user->id);		
2135		}		
2136				
2137		/* ARGSUSED */		
2138		void process_inq_pam_cmd(user)		
2139		USER_INFO *user;		
2140		{		
2141		u_long panno;		
2142		u_long lanesum;		
2143		u_char attrib;		



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

19/36

```

LINE # SOURCE TEXT
2159 DPRINTF(("inside process_inq_pam_cmd\n"));
2160
2161 reset_obuf();
2162 lm_put_int(INQ_PAM_ANS);
2163 laneso = lm_get_int();
2164 panno = lm_get_int();
2165
2166 if ((laneso < 0) || (laneso >= MAX_LANE_COUNT)) {
2167     lm_queue_message(ERROR_MSG, "illegal lane: %d",
2168                     laneso);
2169     end_queue_message();
2170     end_put(user->id);
2171     return;
2172 }
2173
2174 if ((panno < 0) || (panno >= MAX_PAM_COUNT)) {
2175     lm_queue_message(ERROR_MSG, "illegal Fast Pattern Memory: %d",
2176                     panno);
2177     end_queue_message();
2178     end_put(user->id);
2179     return;
2180 }
2181
2182 if (system_config->lane[laneso]->pac_present == FALSE) {
2183     lm_queue_message(ERROR_MSG, "lane: %c does not exist",
2184                     'A' + laneso);
2185     end_queue_message();
2186     end_put(user->id);
2187     return;
2188 }
2189
2190 if (system_config->lane[laneso]->pam[panno] == NULL) {
2191     lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c does not exist",
2192                     panno, 'A' + laneso);
2193     end_queue_message();
2194     end_put(user->id);
2195     return;
2196 }
2197
2198 switch (attrib = lm_get_int()) {
2199     case LM_PAM_MEMORY_SIZE:
2200         lm_queue_message();
2201         lm_put_int(system_config->lane[laneso]->pam[panno]->mem_size);
2202         break;
2203     default:
2204         lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2205                         attrib);
2206         end_queue_message();
2207         break;
2208 }
2209
2210 end_put(user->id);
2211 }
2212
2213 /* ARCUSED */
2214 void process_inq_pel_cmd(user)
2215 USER_INFO *user;
2216 {
2217     DAB_INFO *dab_ptr;
2218     long laneso;
2219     long slotso;
2220     u_char dabso;
2221     u_char unitso;
2222     u_char attrib;
2223
2224     DPRINTF(("inside process_inq_pel_cmd\n"));
2225
2226     reset_obuf();
2227     lm_put_int(INQ_PEL_ANS);
2228
2229     laneso = lm_get_int();
2230     slotso = lm_get_int();
2231     attrib = lm_get_int();
2232
2233     if ((laneso < 0) || (laneso >= MAX_LANE_COUNT)) {
2234         lm_queue_message(ERROR_MSG, "illegal lane: %d",
2235                         laneso);
2236         end_queue_message();
2237         end_put(user->id);
2238         return;
2239     }
2240
2241     if ((slotso < 0) || (slotso >= MAX_SLOT_COUNT)) {
2242         lm_queue_message(ERROR_MSG, "illegal slot: %d",
2243                         slotso);
2244         end_queue_message();
2245         end_put(user->id);
2246         return;
2247     }
2248
2249     if (system_config->lane[laneso]->pel[slotso] == NULL) {
2250         if (attrib == LM_DOES_PEL_EXIST) {
2251             end_queue_message();
2252             lm_put_int(LM_FALSE);
2253         }
2254         else {
2255             lm_queue_message(ERROR_MSG, "Pin Electronics Module in lane: %c slot: %d does not exist",
2256                             'A' + laneso, slotso);
2257             end_queue_message();
2258         }
2259         end_put(user->id);
2260         return;
2261     }
2262
2263     switch (attrib) {
2264         case LM_DOES_PEL_EXIST:
2265             end_queue_message();
2266             lm_put_int(LM_TRUE);
2267             break;
2268         case LM_IS_DAB_PRESENT:
2269             for (dabso = 0; dabso < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabso) {
2270                 dab_ptr = dab_list[dabso];
2271                 if (dab_ptr == NULL)
2272                     continue;
2273                 for (unitso = 0; unitso < dab_ptr->unit_count; ++unitso) {
2274                     if ((dab_ptr->unit_location[unitso].lane_no == laneso) &&

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
20/37

```

2279      (dab_ptr->unit_location[unitno].slot_no == slotno)) {
2280      end_queue_message();
2281      lm_put_int(LM_TRUE);
2282      end_put(user->id);
2283      return;
2284      }
2285      }
2286      }
2287      }
2288      end_queue_message();
2289      lm_put_int(LM_FALSE);
2290      break;
2291      default:
2292      lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2293      attrib);
2294      end_queue_message();
2295      break;
2296      }
2297      }
2298      end_put(user->id);
2299      }
2300      }
2301      /* ARGUSED */
2302      void process_lm_device_list_cmd(user)
2303      USER_INFO *user;
2304      {
2305      {
2306      u_long   dab_count_mark;
2307      u_char   dab_count;
2308      u_char   dabno;
2309      u_char   i;
2310      char     c;
2311      }
2312      DPRINTF(("inside process_lm_device_list_cmd\n"));
2313      reset_cbuf();
2314      lm_put_int(LM_DEVICE_LIST_ANS);
2315      end_queue_message();
2316      }
2317      dab_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
2318      lm_put_int(0); /* dummy dab_count */
2319      }
2320      dab_count = 0;
2321      for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
2322      if (dab_list[dabno] != NULL) {
2323      ++dab_count;
2324      lm_put_int((u_long)dabno);
2325      }
2326      }
2327      }
2328      for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
2329      if (dab_list[dabno] != NULL) {
2330      for (i = 0; (c = dab_list[dabno]->part_name[i]) != '\0'; ++i)
2331      lm_put_char(c);
2332      lm_put_char('\0');
2333      }
2334      }
2335      }
2336      LM_PUT_LONG_AT_MARK(dab_count_mark, lm_global_conn_ptr, (u_long)dab_count);
2337      end_put(user->id);
2338      return;
2339      }
2340      }
2341      /* ARGUSED */
2342      void process_lm_device_name_cmd(user)
2343      USER_INFO *user;
2344      {
2345      DAB_INFO *dab_ptr;
2346      u_long   device_no;
2347      u_char   i;
2348      char     c;
2349      }
2350      DPRINTF(("inside process_lm_device_name_cmd\n"));
2351      reset_cbuf();
2352      lm_put_int(LM_DEVICE_NAME_ANS);
2353      }
2354      device_no = lm_get_int();
2355      if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) {
2356      lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2357      device_no);
2358      end_queue_message();
2359      end_put(user->id);
2360      return;
2361      }
2362      dab_ptr = dab_list[device_no];
2363      if (dab_ptr == NULL) {
2364      lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2365      device_no);
2366      end_queue_message();
2367      end_put(user->id);
2368      return;
2369      }
2370      for (i = 0; c = dab_ptr->part_name[i]; ++i)
2371      lm_put_char(c);
2372      lm_put_char('\0');
2373      }
2374      end_put(user->id);
2375      }
2376      }
2377      /* ARGUSED */
2378      void process_lm_device_cmd(user)
2379      USER_INFO *user;
2380      {
2381      USER_INFO *userx;
2382      DAB_INFO *dab_ptr;
2383      DAB_INFO *temp;
2384      INSTANCE_INFO *instance;
2385      EXTRA_DEVICE_SPEC *extra_def_ptr;
2386      u_long   device_no;
2387      u_long   total_patterns;
2388      u_short  inst_id;
2389      u_char   attrib;
2390      u_char   userno;
2391      }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89

PAGE #

TIME 6:14:38 pm

21/38

LINE #	SOURCE TEXT
2399	DPRINTF(("inside process_lm_device_cmd\n"));
2400	
2401	reset_obuf();
2402	lm_put_int(INQ_DEVICE_ANS);
2403	
2404	device_no = lm_get_int();
2405	if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) {
2406	lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2407	device_no);
2408	end_queue_message();
2409	end_put(user->id);
2410	return;
2411	}
2412	
2413	temp = dab_list[device_no];
2414	if (temp == NULL) {
2415	lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2416	device_no);
2417	end_queue_message();
2418	end_put(user->id);
2419	return;
2420	}
2421	
2422	switch (attrib = lm_get_int()) {
2423	case LM_DEVICE_STATUS:
2424	end_queue_message();
2425	if (temp->used_as_private == TRUE)
2426	lm_put_int(LM_PRIVATE);
2427	else
2428	lm_put_int(LM_PUBLIC);
2429	break;
2430	case LM_NUMBER_OF_DABS:
2431	end_queue_message();
2432	lm_put_int(temp->segment_count);
2433	break;
2434	case LM_NUMBER_OF_ACTIVE_INSTANCES:
2435	end_queue_message();
2436	lm_put_int(temp->act_inst_count);
2437	break;
2438	case LM_NUMBER_OF_ACTIVE_FAULTS:
2439	end_queue_message();
2440	lm_put_int(temp->act_var_count);
2441	break;
2442	case LM_NUMBER_OF_PATTERNS:
2443	total_patterns = 0;
2444	for (userno = 0; userno < MAX_USER_COUNT; ++userno) {
2445	userx = user_info_array[userno];
2446	
2447	if (userx->active == FALSE)
2448	continue;
2449	
2450	for (inst_id = 0; inst_id < userx->inst_table_size; ++inst_id) {
2451	instance = userx->instance[inst_id];
2452	
2453	if (BOGUS_INSTANCE(userx, instance))
2454	continue;
2455	
2456	extra_def_ptr = (EXTRA_DEVICE_SPEC *)
2457	instance->definition->extra_data;
2458	
2459	if (extra_def_ptr->dab_ok == FALSE)
2460	continue;
2461	
2462	dab_ptr = dab_list[instance->dab_info_index];
2463	
2464	/* If this instance is using the device we are looking for then
2465	* count the pattern usage.
2466	*/
2467	if (dab_ptr == temp) {
2468	if (instance->is_fault == FALSE) {
2469	total_patterns += instance->pattern_count;
2470	}
2471	else {
2472	total_patterns += instance->pattern_count -
2473	instance->common_pattern_count;
2474	}
2475	}
2476	
2477	end_queue_message();
2478	lm_put_int(total_patterns);
2479	break;
2480	default:
2481	lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2482	attrib);
2483	end_queue_message();
2484	break;
2485	
2486	end_put(user->id);
2487	
2488	
2489	
2490	
2491	/* ARGSUSED */
2492	void process_lm_dab_cmd(user)
2493	USER_INFO *user;
2494	{
2495	DAB_INFO *temp;
2496	SEGMENT_EL *seg_ptr;
2497	u_long device_no;
2498	long dab_number;
2499	char str[16];
2500	u_char i;
2501	u_char attrib;
2502	char c;
2503	
2504	DPRINTF(("inside process_lm_dab_cmd\n"));
2505	
2506	reset_obuf();
2507	lm_put_int(INQ_DAB_ANS);
2508	
2509	device_no = lm_get_int();
2510	if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) {
2511	lm_queue_message(ERROR_MSG, "invalid device number: %d specified",
2512	device_no);
2513	end_queue_message();
2514	end_put(user->id);
2515	return;
2516	}
2517	
2518	temp = dab_list[device_no];

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89	PAGE # 22/39
LINE #		SOURCE TEXT		
2519		if (temp == NULL) {		
2520		lm_queue_message(ERROR_MSG, "invalid device number: %d specified",		
2521		device_no);		
2522		end_queue_message();		
2523		end_put(user->id);		
2524		return;		
2525		}		
2526		dab_number = lm_get_int();		
2527		if (dab_number >= temp->segment_count) {		
2528		lm_queue_message(ERROR_MSG, "illegal Device Adapter number: %d",		
2529		dab_number);		
2530		end_queue_message();		
2531		end_put(user->id);		
2532		return;		
2533		}		
2534		if (temp->segment[0] != NULL)		
2535		seg_ptr = temp->segment[0];		
2536		else {		
2537		seg_ptr = temp->segment[dab_number + 1];		
2538		if (seg_ptr == NULL) {		
2539		lm_queue_message(ERROR_MSG, "internal error: segment D/S not found");		
2540		end_queue_message();		
2541		end_put(user->id);		
2542		return;		
2543		}		
2544		switch (attrib = lm_get_int()) {		
2545		case LM_DAB_TYPE:		
2546		end_queue_message();		
2547		for (i = 0; i = seg_ptr->dab_type[i]; ++i)		
2548		lm_put_char(c);		
2549		lm_put_char('\0');		
2550		break;		
2551		case LM_DAB_REVISION:		
2552		end_queue_message();		
2553		for (i = 0; i = seg_ptr->revision[i]; ++i)		
2554		lm_put_char(c);		
2555		lm_put_char('\0');		
2556		break;		
2557		case LM_DAB_MANUFACTURER:		
2558		end_queue_message();		
2559		for (i = 0; i = seg_ptr->man_id[i]; ++i)		
2560		lm_put_char(c);		
2561		lm_put_char('\0');		
2562		break;		
2563		case LM_DAB_MODEL:		
2564		end_queue_message();		
2565		for (i = 0; i = seg_ptr->model_maker[i]; ++i)		
2566		lm_put_char(c);		
2567		lm_put_char('\0');		
2568		break;		
2569		case LM_DAB_MODEL_REVISION:		
2570		end_queue_message();		
2571		for (i = 0; i = seg_ptr->model_revision[i]; ++i)		
2572		lm_put_char(c);		
2573		lm_put_char('\0');		
2574		break;		
2575		case LM_DAB_INSERTIONS:		
2576		end_queue_message();		
2577		(void)sprintf(str, "%d", seg_ptr->insertion_count);		
2578		for (i = 0; i = str[i]; ++i)		
2579		lm_put_char(c);		
2580		lm_put_char('\0');		
2581		break;		
2582		default:		
2583		lm_queue_message(ERROR_MSG, "illegal attribute: %d",		
2584		attrib);		
2585		end_queue_message();		
2586		break;		
2587		}		
2588		end_put(user->id);		
2589		}		
2590		/* ABCUSED */		
2591		void process_lm_dab_loc_cmd(user)		
2592		USER_INFO *user;		
2593		{		
2594		DAB_INFO *temp;		
2595		u_long device_no;		
2596		u_long dab_no;		
2597		DPRINTF(("inside process_lm_dab_loc_cmd\n"));		
2598		reset_obuf();		
2599		lm_put_int(ING_DAB_LOC_AMS);		
2600		device_no = lm_get_int();		
2601		dab_no = lm_get_int();		
2602		if (device_no >= (MAX_LANE_COUNT * MAX_SLOT_COUNT)) {		
2603		lm_queue_message(ERROR_MSG, "invalid device number: %d specified",		
2604		device_no);		
2605		end_queue_message();		
2606		end_put(user->id);		
2607		return;		
2608		}		
2609		temp = dab_list[device_no];		
2610		if (temp == NULL) {		
2611		lm_queue_message(ERROR_MSG, "invalid device number: %d specified",		
2612		device_no);		
2613		end_queue_message();		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
23/40

```

LINE #          SOURCE TEXT
2639      end_put(user->id);
2640      return;
2641  }
2642
2643      if (temp->segment[0] != NULL) {
2644          /* Single segment use */
2645          if (dab_no != 0) {
2646              lm_queue_message(ERROR_MSG, "invalid Device Adapter number: %d specified",
2647                              dab_no);
2648              end_queue_message();
2649              end_put(user->id);
2650              return;
2651          }
2652      }
2653      else {
2654          /* Multi segment Device */
2655          if (dab_no != temp->segment_count) {
2656              lm_queue_message(ERROR_MSG, "invalid Device Adapter number: %d specified",
2657                              dab_no);
2658              end_queue_message();
2659              end_put(user->id);
2660              return;
2661          }
2662      }
2663
2664      end_queue_message();
2665
2666      dab_loc(temp, dab_no);
2667
2668      end_put(user->id);
2669  }
2670
2671  void process_inq_instance_cmd(user)
2672  USER_INFO *user;
2673  {
2674      INSTANCE_INFO *instance;
2675      u_long      freq;
2676      u_short     inst_id;
2677      u_short     attrib;
2678
2679      DPRINTF(("inside process_inq_instance_cmd\n"));
2680
2681      reset_obuf();
2682      lm_put_int(INQ_INSTANCE_ANS);
2683
2684      inst_id = lm_get_short();
2685      attrib = lm_get_int();
2686
2687      /* verify inst_id */
2688      if (inst_id >= user->inst_table_size) {
2689          lm_queue_message(ERROR_MSG, "invalid instance id: %d specified",
2690                          inst_id);
2691          end_queue_message();
2692          end_put(user->id);
2693          return;
2694      }
2695
2696      instance = user->instance[inst_id];
2697      if (BOGUS_INSTANCE(user, instance)) {
2698          lm_queue_message(ERROR_MSG, "invalid instance id: %d specified",
2699                          inst_id);
2700          end_queue_message();
2701          end_put(user->id);
2702          return;
2703      }
2704
2705      if (instance->is_fault) {
2706          lm_queue_message(ERROR_MSG, "instance id: %d is a fault",
2707                          inst_id);
2708          end_queue_message();
2709          end_put(user->id);
2710          return;
2711      }
2712
2713      switch (attrib) {
2714      case LM_NUMBER_OF_PATTERNS:
2715          end_queue_message();
2716          lm_put_int(instance->pattern_count);
2717          break;
2718      case LM_PATTERN_FREQUENCY:
2719          end_queue_message();
2720          freq = (double)1000000000000.0 / (double)((EXTRA_DEVICE_SPEC *)
2721              (instance->definition->extra_data))->actual_phy_clock_period + 0.5;
2722          lm_put_int(freq);
2723          break;
2724      default:
2725          lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2726                          attrib);
2727          end_queue_message();
2728          break;
2729      }
2730
2731      end_put(user->id);
2732  }
2733
2734  void process_inq_fault_cmd(user)
2735  USER_INFO *user;
2736  {
2737      INSTANCE_INFO *fault;
2738      u_long      freq;
2739      u_short     fault_id;
2740      u_short     attrib;
2741
2742      DPRINTF(("inside process_inq_fault_cmd\n"));
2743
2744      reset_obuf();
2745      lm_put_int(INQ_FAULT_ANS);
2746
2747      fault_id = lm_get_short();
2748      attrib = lm_get_int();
2749
2750      /* verify fault_id */
2751      if (fault_id >= user->inst_table_size) {
2752          lm_queue_message(ERROR_MSG, "invalid fault id: %d specified",
2753                          fault_id);
2754          end_queue_message();
2755          end_put(user->id);
2756          return;
2757      }
2758  }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89	PAGE # 24/41
LINE #	SOURCE TEXT			
2759	fault = user->instance[fault_id];			
2760	if (BOGUS_INSTANCE(user, fault)) {			
2761	lm_queue_message(ERROR_MSG, "invalid fault id: %d specified",			
2762	fault_id);			
2763	end_queue_message();			
2764	end_put(user->id);			
2765	return;			
2766	}			
2767				
2768	if (! fault->is_fault) {			
2769	lm_queue_message(ERROR_MSG, "fault id: %d is an instance",			
2770	fault_id);			
2771	end_queue_message();			
2772	end_put(user->id);			
2773	return;			
2774	}			
2775				
2776	switch (attrib) {			
2777	case LM_NUMBER_OF_PATTERNS:			
2778	end_queue_message();			
2779	lm_put_inst(fault->patterns_count);			
2780	break;			
2781	case LM_PATTERN_FREQUENCY:			
2782	end_queue_message();			
2783	freq = (double)1000000000000.0 / ((double)((EXTRA_DEVICE_SPEC *)			
2784	(fault->definition->extra_data))->actual_phy_clock_period + 0.5;			
2785	lm_put_inst(freq);			
2786	break;			
2787	default:			
2788	lm_queue_message(ERROR_MSG, "illegal attribute: %d",			
2789	attrib);			
2790	end_queue_message();			
2791	break;			
2792	}			
2793				
2794	end_put(user->id);			
2795				
2796				
2797	void process_img_avail_ptrn_cmd(user)			
2798	USER_INFO *user;			
2799	{			
2800	DEVICE_SPEC *def_ptr;			
2801	DAB_INFO *dab_ptr;			
2802	long temp;			
2803	long min_lane_ptrn_count;			
2804	u_short def_id;			
2805	char dab_info_index;			
2806	u_char lanes;			
2807				
2808	/* Calculate the maximum number of patterns this definition can have.			
2809	* This is used to RESTORE to first check if we can allocate the patterns			
2810	* before actually sending the patterns.			
2811	*/			
2812	DPRINTF(("inside process_img_avail_ptrn_cmd\n"));			
2813				
2814	reset_chan();			
2815	lm_put_inst(IMG_AVAIL_PTRN_ANS);			
2816				
2817	def_id = lm_get_abort();			
2818				
2819				
2820	/* verify def_id */			
2821	if (def_id > user->def_table_size) {			
2822	lm_queue_message(ERROR_MSG, "invalid definition id: %d specified",			
2823	def_id);			
2824	end_queue_message();			
2825	end_put(user->id);			
2826	return;			
2827	}			
2828				
2829	def_ptr = user->definition[def_id];			
2830	if (BOGUS_DEFINITION(user, def_ptr)) {			
2831	lm_queue_message(ERROR_MSG, "invalid definition id: %d specified",			
2832	def_id);			
2833	end_queue_message();			
2834	end_put(user->id);			
2835	return;			
2836	}			
2837				
2838	if ((short)(dab_info_index = find_dab(dab_list, def_ptr->device_name,			
2839	(u_char)def_ptr->device_type)) == -1) {			
2840	lm_queue_message(ERROR_MSG, "device name: %s not found or being used as PRIVATE",			
2841	def_ptr->device_name);			
2842	end_queue_message();			
2843	end_put(user->id);			
2844	return;			
2845	}			
2846				
2847	dab_ptr = dab_list[dab_info_index];			
2848				
2849	min_lane_ptrn_count = MAXINT;			
2850				
2851	for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) {			
2852				
2853	if (dab_ptr->lane_used[laneno] == FALSE)			
2854	continue;			
2855				
2856	temp = count_avail_patterns((u_char)laneno) * PTRN_PER_BLOCK;			
2857				
2858	if (temp < min_lane_ptrn_count)			
2859	min_lane_ptrn_count = temp;			
2860				
2861	}			
2862				
2863	end_queue_message();			
2864				
2865	lm_put_inst(min_lane_ptrn_count);			
2866				
2867	end_put(user->id);			
2868				
2869				
2870	void process_tmeasurement_cmd(user)			
2871	USER_INFO *user;			
2872	{			
2873	DEVICE_SPEC *def_ptr;			
2874	INSTANCE_INFO *instance;			
2875	u_short inst_id;			
2876	u_char on_or_off;			
2877				
2878	DPRINTF(("inside process_tmeasurement_cmd\n"));			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	25/42

```

LINE # SOURCE TEXT
2879
2880 reset_obuf();
2881 lm_put_int(TMEASUREMENT_ANS);
2882
2883 inst_id = lm_get_short();
2884 on_or_off = lm_get_char();
2885
2886 /* verify inst_id */
2887 if (inst_id >= user->inst_table_size) {
2888     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
2889         inst_id);
2890     end_queue_message();
2891     end_put(user->id);
2892     return;
2893 }
2894
2895 instance = user->instance[inst_id];
2896 if (BOGUS_INSTANCE(user, instance)) {
2897     lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: %d specified",
2898         inst_id);
2899     end_queue_message();
2900     end_put(user->id);
2901     return;
2902 }
2903
2904 switch (on_or_off) {
2905     case LM_TIMING_OFF:
2906         instance->enable_timing_mes = FALSE;
2907         break;
2908     case LM_TIMING_ON:
2909         def_ptr = instance->definition;
2910         if (def_ptr->tbl_tie_measure == TRUE) {
2911             lm_queue_message(ERROR_MSG, "Timing Measurement is inhibited for device: %s\n",
2912                 def_ptr->device_name);
2913         }
2914         else {
2915             if (def_ptr->device_type == PUBLIC)
2916                 instance->enable_timing_mes = TRUE;
2917             else {
2918                 if ((instance->has_history == TRUE) &&
2919                     (instance->purge_ptrn_on_next_eval == FALSE)) {
2920                     instance->enable_timing_mes = TRUE;
2921                 }
2922                 else {
2923                     lm_queue_message(ERROR_MSG, "private instance: %s does not have history",
2924                         instance->device_info_string);
2925                 }
2926             }
2927         }
2928         break;
2929     default:
2930         lm_queue_message(ERROR_MSG, "illegal attribute: %d",
2931             on_or_off);
2932         break;
2933 }
2934
2935 end_queue_message();
2936 end_put(user->id);
2937
2938 void process_loop_ptrn_cmd(user)
2939 USER_INFO *user;
2940 {
2941     INSTANCE_INFO *instance;
2942     DEVICE_SPEC *def_ptr;
2943     u_short inst_id;
2944     extern char *modeler_state;
2945
2946     DPRINTF(("inside process_loop_ptrn_cmd\n"));
2947
2948     reset_obuf();
2949     lm_put_int(LOOP_PTRN_ANS);
2950
2951     inst_id = lm_get_short();
2952
2953     /* verify inst_id */
2954     if (inst_id >= user->inst_table_size) {
2955         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
2956             inst_id);
2957         end_queue_message();
2958         end_put(user->id);
2959         return;
2960     }
2961
2962     instance = user->instance[inst_id];
2963     if (BOGUS_INSTANCE(user, instance)) {
2964         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
2965             inst_id);
2966         end_queue_message();
2967         end_put(user->id);
2968         return;
2969     }
2970
2971     def_ptr = instance->definition;
2972     if (def_ptr == NULL) {
2973         lm_queue_message(ERROR_MSG, "internal error: no definition for instance");
2974         end_queue_message();
2975         end_put(user->id);
2976         return;
2977     }
2978
2979     if (((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok == FALSE) {
2980         lm_queue_message(ERROR_MSG, "Device Adapter was Removed");
2981         end_queue_message();
2982         end_put(user->id);
2983         return;
2984     }
2985
2986     if (instance->failed_to_alloc_ptrn == TRUE) {
2987         lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory; cannot continue simulation");
2988         end_queue_message();
2989         end_put(user->id);
2990         return;
2991     }
2992
2993     if (instance->fatal_error == TRUE) {
2994         lm_queue_message(ERROR_MSG, "fatal error encountered on this instance; cannot continue simulation");
2995         end_queue_message();
2996         end_put(user->id);
2997     }
2998

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	26/43

```

LINE #          SOURCE TEXT
2999      )
3000      return;
3001
3002      if (instance->definition->device_type == PRIVATE) {
3003          lm_queue_message(ERROR_MSG, "Cannot loop pattern for PRIVATE device");
3004          end_queue_message();
3005          end_put(user->id);
3006          return;
3007      }
3008
3009      if (active_user_count() > 1) {
3010          lm_queue_message(ERROR_MSG, "modeler busy");
3011          end_queue_message();
3012          end_put(user->id);
3013          return;
3014      }
3015
3016      modeler_state = RUNNING_LOOPMODE;
3017      end_queue_message();
3018      end_put(user->id); /* ack now */
3019
3020      (void)do_loop_ptr(instance);
3021      modeler_state = MODELER_RUNNING;
3022  }
3023
3024  active_user_count()
3025  {
3026      int i, count;
3027
3028      for (i = count = 0; i < MAX_USER_COUNT; ++i)
3029          if (user_info_array[i]->active == TRUE)
3030              ++count;
3031
3032      return(count);
3033  }
3034
3035  void process_reset_inst_cmd(user)
3036  USER_INFO *user;
3037  {
3038      DEVICE_SPEC *def_ptr;
3039      INSTANCE_INFO *instance;
3040      DAB_INFO *dab_ptr;
3041      long inst_id;
3042      u_short error_count;
3043      u_short warning_count;
3044
3045      DPRINTF(("inside process_reset_inst_cmd\n"));
3046
3047      reset_cmd();
3048      lm_put_inst(RESET_INST_AWS);
3049
3050      inst_id = lm_get_short();
3051
3052      /* verify the inst_id */
3053      if ((inst_id < 0) || (inst_id >= user->inst_table_size)) {
3054          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified");
3055          end_queue_message();
3056          end_put(user->id);
3057          return;
3058      }
3059
3060      instance = user->instances[inst_id];
3061      if (ISOCUS_INSTANCE(user, instance)) {
3062          lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance id: id specified");
3063          end_queue_message();
3064          end_put(user->id);
3065          return;
3066      }
3067
3068      def_ptr = instance->definition;
3069      if (((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok == FALSE) {
3070          lm_queue_message(ERROR_MSG, "Device Adapter was removed");
3071          end_queue_message();
3072          end_put(user->id);
3073          return;
3074      }
3075
3076      if (instance->is_fault) {
3077          lm_queue_message(ERROR_MSG, "internal simulator error: cannot reset a fault; fault id: id");
3078          end_queue_message();
3079          end_put(user->id);
3080          return;
3081      }
3082
3083      /* Check if this instance still has faults */
3084      /* ??? Can't do this with the current structure.
3085       * Just make sure this condition is checked on the host.
3086       */
3087
3088      return_all_ptr_block(instance);
3089      dab_ptr = dab_list(instance->dab_info_index);
3090      reinitialize_instance(instance);
3091      build_static_pattern_seq(instance);
3092      lm_message_types(error_count, warning_count);
3093
3094      if (error_count != 0) {
3095          dab_ptr->used_as_private = FALSE;
3096          return_all_ptr_block(instance); /*
3097          end_queue_message();
3098          */
3099      }
3100
3101      else {
3102          if (def_ptr->device_type == PRIVATE) {
3103              instance->has_history = TRUE;
3104              instance->purge_ptrn_on_next_eval = TRUE;
3105          }
3106          end_queue_message();
3107      }
3108
3109      copy_initial_values(instance);
3110

```



```

LINE # SOURCE TEXT
3119 }
3120
3121 end_put(user->fd);
3122 }
3123
3124 /* ARGSUSED */
3125 void process_no_such(user)
3126 USER_INFO *user;
3127 {
3128     DPRINTF(("inside process_no_such\n"));
3129     reset_obuf();
3130
3131     lm_put_int(NO_SUCH_ANS);
3132     end_put(user->fd);
3133 }
3134
3135 void process_test_network_cmd(user)
3136 USER_INFO *user;
3137 {
3138     USER_INFO *userx;
3139     u_long j;
3140     u_long total_number;
3141     u_long checksum;
3142     u_long checksum2;
3143     u_long command;
3144     u_long subcommand;
3145     u_long usernr;
3146     char error_string[512];
3147     u_short temp;
3148     char c;
3149
3150     DPRINTF(("inside process_test_network\n"));
3151     reset_obuf();
3152
3153     lm_put_int(TEST_NETWORK_ANS);
3154
3155     command = lm_get_int();
3156     switch (command) {
3157     case 1:
3158         total_number = lm_get_int();
3159         DPRINTF(("total_number received = %d\n", total_number));
3160         checksum = 0;
3161         for (j = 0; j < total_number; ++j) {
3162             checksum += lm_get_int();
3163         }
3164         DPRINTF(("checksum on command: %d\n", checksum));
3165         if ((checksum2 = lm_get_int()) != checksum) {
3166             DPRINTF(("Checksum error\n"));
3167             lm_queue_message(ERROR_MSG, "internal error: checksum error on command: exp: %08x got: %08x",
3168                             checksum2, checksum);
3169             end_queue_message();
3170             end_put(user->fd);
3171             return;
3172         }
3173         end_queue_message();
3174         lm_put_int(total_number);
3175         checksum = 0;
3176         for (j = 0; j < total_number; ++j) {
3177             lm_put_int(j*123);
3178             checksum += j * 123;
3179         }
3180         DPRINTF(("checksum on answer: %d\n", checksum));
3181         lm_put_int(checksum);
3182         break;
3183     case 2:
3184         subcommand = lm_get_int();
3185         if (subcommand == 1000) {
3186             /* lock command */
3187             if (modeler_is_locked == TRUE) {
3188                 lm_queue_message(ERROR_MSG, "modeler is currently locked by: %s",
3189                                 user_with_lock);
3190                 end_queue_message();
3191                 end_put(user->fd);
3192                 return;
3193             }
3194             else {
3195                 for (j = 0; (c = lm_get_char()) != '\0'; ++j)
3196                     user_with_lock[j] = c;
3197                 user_with_lock[j] = '\0';
3198                 modeler_is_locked = TRUE;
3199             }
3200         }
3201         else if (subcommand == 2000) {
3202             /* unlock command */
3203             if (modeler_is_locked == TRUE) {
3204                 if (strcmp(user_with_lock, user->username) != 0) {
3205                     lm_queue_message(ERROR_MSG, "failed to unlock modeler, user: %s holds the lock",
3206                                     user_with_lock);
3207                     end_queue_message();
3208                     end_put(user->fd);
3209                     return;
3210                 }
3211                 modeler_is_locked = FALSE;
3212             }
3213         }
3214         else {
3215             lm_queue_message(ERROR_MSG, "unknown lock/unlock command");
3216             end_queue_message();
3217             end_put(user->fd);
3218             return;
3219         }
3220         break;
3221     case 3:
3222         usernr = lm_get_int();
3223         if (usernr >= MAX_USER_COUNT) {
3224             lm_queue_message(ERROR_MSG, "user: %d too large", usernr);
3225             end_queue_message();
3226         }
3227     }
3228 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c	DATE 5/23/89	PAGE # 28/45
LINE #	SOURCE TEXT			
3239	end_put(user->fd);			
3240	return;			
3241	}			
3242	userx = ....._info_array(userno);			
3243				
3244				
3245	if (userx == NULL) {			
3246	lm_queue_message(ERROR_MSG, "user number: %d does not exist", userno);			
3247	end_queue_message();			
3248	end_put(user->fd);			
3249	return;			
3250	}			
3251				
3252	abort_user(userx);			
3253				
3254	end_queue_message();			
3255				
3256	end_put(user->fd);			
3257				
3258	if (set_close_connection_for_server(table_of_coans[userno]) != SUCCESS) {			
3259	while (1) {			
3260	if (lm_dequeue_message(&temp, error_string) != FAILURE)			
3261	DPRINTF(("ts", error_string));			
3262	else			
3263	break;			
3264	}			
3265	}			
3266	return;			
3267	case 4:			
3268	/* Clear profile counter */			
3269	profile_clear();			
3270	break;			
3271	case 5:			
3272	/* Dump profile information */			
3273	end_queue_message();			
3274	profile_dump_count();			
3275	break;			
3276	default:			
3277	lm_queue_message(ERROR_MSG, "unknown test network command");			
3278	end_queue_message();			
3279	break;			
3280	}			
3281	end_put(user->fd);			
3282	return;			
3283	}			
3284				
3285	void process_abort_cmd(user)			
3286	USER_INFO *user;			
3287	{			
3288	u_abort temp;			
3289	char error_string[512];			
3290				
3291	DPRINTF(("inside process_abort_cmd\n"));			
3292				
3293	reset_obuf();			
3294	lm_put_int(ABORT_ANS);			
3295				
3296	abort_user(user);			
3297				
3298	end_queue_message();			
3299				
3300	DPRINTF(("exiting process_abort_cmd; user: %d\n", lm_global_conn_ptr->fd));			
3301				
3302	end_put(user->fd);			
3303				
3304				
3305	if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {			
3306	while (1) {			
3307	if (lm_dequeue_message(&temp, error_string) != FAILURE)			
3308	DPRINTF(("ts", error_string));			
3309	else			
3310	break;			
3311	}			
3312	}			
3313	}			
3314				
3315	/* ARGSUSED */			
3316	void process_label_dab_cmd(user)			
3317	USER_INFO *user;			
3318	{			
3319	DAB_EEPROM dab_eprom;			
3320	u_long configuration;			
3321	u_char lanes;			
3322	u_char slotno;			
3323	char dab_type[MAX_STRING_LENGTH];			
3324	char device_name[MAX_STRING_LENGTH];			
3325	char model_maker[MAX_STRING_LENGTH];			
3326	char model_revision[MAX_STRING_LENGTH];			
3327	char manufacturer[MAX_STRING_LENGTH];			
3328	char revision[MAX_STRING_LENGTH];			
3329	u_char i;			
3330	u_char j;			
3331	u_char c;			
3332	u_char lanes_high;			
3333	u_char slots_wide;			
3334	u_char segment_number;			
3335	DAB_INFO *dab_ptr;			
3336	u_char dabno;			
3337	u_char unitno;			
3338				
3339	DPRINTF(("inside process_label_dab_cmd\n"));			
3340				
3341	reset_obuf();			
3342	lm_put_int(LABEL_DAB_ANS);			
3343				
3344	lanes = lm_get_char();			
3345	if (lanes > MAX_LANE_COUNT) {			
3346	lm_queue_message(ERROR_MSG, "illegal lane: %d", lanes);			
3347	end_queue_message();			
3348	end_put(user->fd);			
3349	return;			
3350				
3351	}			
3352				
3353	slotno = lm_get_char();			
3354	if (slotno > MAX_SLOT_COUNT) {			
3355	lm_queue_message(ERROR_MSG, "illegal slot: %d", slotno);			
3356	end_queue_message();			
3357	end_put(user->fd);			
3358				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	29/46

LINE #	SOURCE TEXT
3359	return;
3360	}
3361	/* dab_type */
3362	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3363	dab_type[i] = c;
3364	dab_type[i] = '\0';
3365	
3366	if (strlen(dab_type) > TYPE_LENGTH) {
3367	lm_queue_message(ERROR_MSG, "Device Adapter type: %s too long; max allowed: %d chars",
3368	dab_type, TYPE_LENGTH);
3369	end_queue_message();
3370	end_put(user->id);
3371	return;
3372	}
3373	
3374	/* device_name */
3375	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3376	device_name[i] = c;
3377	device_name[i] = '\0';
3378	
3379	if (strlen(device_name) > NAME_LENGTH) {
3380	lm_queue_message(ERROR_MSG, "device_name: %s too long; max allowed: %d chars",
3381	device_name, NAME_LENGTH);
3382	end_queue_message();
3383	end_put(user->id);
3384	return;
3385	}
3386	
3387	/* model_maker */
3388	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3389	model_maker[i] = c;
3390	model_maker[i] = '\0';
3391	
3392	if (strlen(model_maker) > MAKER_LENGTH) {
3393	lm_queue_message(ERROR_MSG, "Device Adapter model_maker: %s too long; max allowed: %d chars",
3394	model_maker, MAKER_LENGTH);
3395	end_queue_message();
3396	end_put(user->id);
3397	return;
3398	}
3399	
3400	/* model_revision */
3401	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3402	model_revision[i] = c;
3403	model_revision[i] = '\0';
3404	
3405	if (strlen(model_revision) > REV_LENGTH) {
3406	lm_queue_message(ERROR_MSG, "Device Adapter model_revision: %s too long; max allowed: %d chars",
3407	model_revision, REV_LENGTH);
3408	end_queue_message();
3409	end_put(user->id);
3410	return;
3411	}
3412	
3413	/* manufacturer */
3414	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3415	manufacturer[i] = c;
3416	manufacturer[i] = '\0';
3417	
3418	if (strlen(manufacturer) > MAN_LENGTH) {
3419	lm_queue_message(ERROR_MSG, "Device Adapter manufacturer: %s too long; max allowed: %d chars",
3420	manufacturer, MAN_LENGTH);
3421	end_queue_message();
3422	end_put(user->id);
3423	return;
3424	}
3425	
3426	/* revision */
3427	for (i = 0; ((c = lm_get_char()) != '\0'); ++i)
3428	revision[i] = c;
3429	revision[i] = '\0';
3430	
3431	if (strlen(revision) > REV_LENGTH) {
3432	lm_queue_message(ERROR_MSG, "Device Adapter revision: %s too long; max allowed: %d chars",
3433	revision, REV_LENGTH);
3434	end_queue_message();
3435	end_put(user->id);
3436	return;
3437	}
3438	
3439	lanes_high = lm_get_char();
3440	if (lanes_high > MAX_LANE_COUNT) {
3441	lm_queue_message(ERROR_MSG, "illegal lanes_high: %d; max allowed: %d",
3442	lanes_high, MAX_LANE_COUNT);
3443	end_queue_message();
3444	end_put(user->id);
3445	return;
3446	}
3447	
3448	if (lanes_high == 0) {
3449	lm_queue_message(ERROR_MSG, "illegal lanes_high: 0; min allowed: 1");
3450	end_queue_message();
3451	end_put(user->id);
3452	return;
3453	}
3454	
3455	slots_wide = lm_get_char();
3456	if (slots_wide > MAX_SLOT_COUNT) {
3457	lm_queue_message(ERROR_MSG, "invalid slots_wide: %d; max allowed: %d",
3458	slots_wide, MAX_SLOT_COUNT);
3459	end_queue_message();
3460	end_put(user->id);
3461	return;
3462	}
3463	
3464	if (slots_wide == 0) {
3465	lm_queue_message(ERROR_MSG, "invalid slots_wide: 0; min allowed: 1");
3466	end_queue_message();
3467	end_put(user->id);
3468	return;
3469	}
3470	
3471	segment_number = lm_get_char();
3472	if (segment_number > MAX_SEGMENT_PER_DEVICE) {
3473	lm_queue_message(ERROR_MSG, "invalid segment_number: %d; max allowed: %d",
3474	segment_number, MAX_SEGMENT_PER_DEVICE);
3475	end_queue_message();
3476	end_put(user->id);
3477	return;
3478	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE 5/23/89  
TIME 6:14:38 pm

PAGE #  
30/47

```

LINE # SOURCE TEXT
3479 }
3480
3481 if (check_dab_present(laneno, slotno) == FAILURE) {
3482     lm_queue_message(ERROR_MSG, "no Device Adapter in lane: %c slot: %d",
3483         'A' + laneno, slotno);
3484     end_queue_message();
3485     end_put(user->fd);
3486     return;
3487 }
3488
3489 /* Check through the dab list and see if any active instances or faults
3490 exit for this position.
3491 */
3492 for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {
3493     dab_ptr = dab_list[dabno];
3494     if (dab_ptr == NULL)
3495         continue;
3496
3497     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
3498         if ((dab_ptr->unit_location[unitno].lane_no == laneno) &&
3499             (dab_ptr->unit_location[unitno].slot_no == slotno)) {
3500             if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) > 0) {
3501                 lm_queue_message(ERROR_MSG, "Device Adapter in lane: %c slot: %d is in use.",
3502                     'A' + laneno, slotno);
3503                 end_queue_message();
3504                 end_put(user->fd);
3505                 return;
3506             }
3507         }
3508     }
3509 }
3510
3511 /* Get the insertion count into the dab_eeprom structure */
3512 (void)lm_read_eeprom(laneno * MAX_SLOT_COUNT + slotno, &dab_eeprom);
3513
3514 for (i = 0; device_name[i] != '\0'; ++i)
3515     dab_eeprom.device_name[i] = device_name[i];
3516
3517 for (; i < NAME_LENGTH; ++i)
3518     dab_eeprom.device_name[i] = '\0';
3519
3520 for (i = 0; model_maker[i] != '\0'; ++i)
3521     dab_eeprom.model_maker[i] = model_maker[i];
3522
3523 for (; i < MAKER_LENGTH; ++i)
3524     dab_eeprom.model_maker[i] = '\0';
3525
3526 for (i = 0; model_revision[i] != '\0'; ++i)
3527     dab_eeprom.model_revision[i] = model_revision[i];
3528
3529 for (; i < REV_LENGTH; ++i)
3530     dab_eeprom.model_revision[i] = '\0';
3531
3532 for (i = 0; manufacturer[i] != '\0'; ++i)
3533     dab_eeprom.dab_manufacturer[i] = manufacturer[i];
3534
3535 for (; i < MAN_LENGTH; ++i)
3536     dab_eeprom.dab_manufacturer[i] = '\0';
3537
3538 for (i = 0; revision[i] != '\0'; ++i)
3539     dab_eeprom.dab_revision[i] = revision[i];
3540
3541 for (; i < REV_LENGTH; ++i)
3542     dab_eeprom.dab_revision[i] = '\0';
3543
3544 for (i = 0; dab_type[i] != '\0'; ++i)
3545     dab_eeprom.dab_type[i] = dab_type[i];
3546
3547 for (; i < TYPE_LENGTH; ++i)
3548     dab_eeprom.dab_type[i] = '\0';
3549
3550 dab_eeprom.segment_number = segment_number;
3551
3552 configuration = 0;
3553 for (i = 0; i < lanes_high; ++i) {
3554     for (j = 0; j < slots_wide; ++j) {
3555         configuration |= 1 << (i * MAX_SLOT_COUNT + j);
3556     }
3557 }
3558
3559 dab_eeprom.configuration = configuration;
3560
3561 /* Write the DAB EEPROM structure to the EEPROM on the DAB */
3562 (void)lm_write_eeprom(laneno * MAX_SLOT_COUNT + slotno, &dab_eeprom);
3563
3564 reconfigure_dab();
3565 end_queue_message();
3566 end_put(user->fd);
3567 }
3568
3569 void process_eval_control_cmd(user)
3570 USER_INFO *user;
3571 {
3572     INSTANCE_INFO *instance;
3573     u_short inst_id;
3574     u_short attrib;
3575     long value;
3576
3577     DPRINTF(("inside process_eval_control_cmd\n"));
3578
3579     reset_obuf();
3580     lm_put_inst(EVAL_CONTROL_ANS);
3581
3582     inst_id = lm_get_short();
3583     attrib = lm_get_inst();
3584     value = lm_get_inst();
3585
3586     /* verify inst_id */
3587     if (inst_id >= user->inst_table_size) {
3588         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
3589             inst_id);
3590         end_queue_message();
3591         end_put(user->fd);
3592         return;
3593     }
3594
3595     instance = user->instance[inst_id];
3596     if (BUGUS_INSTANCE(user, instance)) {
3597         lm_queue_message(ERROR_MSG, "internal simulator error: invalid instance/fault id: %d specified",
3598             inst_id);

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/function.c

DATE	5/23/89	PAGE #
TIME	6:14:38 pm	31/48

```

LINE #          SOURCE TEXT
3599             inst_id);
3600             end_queue_message();
3601             end_put(user->id);
3602             return;
3603         }
3604
3605         switch (attrib) {
3606             case LM_NUMBER_OF_SAMPLES:
3607                 instance->sample_count = value;
3608                 break;
3609             case LM_NUMBER_OF_EVALUATIONS:
3610                 if (value == LM_EVERY_EVALUATION)
3611                     instance->evaluation_count = MAXINT;
3612                 else
3613                     instance->evaluation_count = value;
3614                 break;
3615             default:
3616                 lm_queue_message(ERROR_MSG, "illegal attribute: id",
3617                                 attrib);
3618                 break;
3619         }
3620
3621         end_queue_message();
3622         end_put(user->id);
3623     }
3624
3625 void process_reboot_cmd(user)
3626 USER_INFO *user;
3627 {
3628     DPRINTF(("inside process_reboot_cmd\n"));
3629
3630     reset_obuf();
3631     lm_put_int(REBOOT_ANS);
3632
3633     #ifdef MODELER
3634     if (check_password(user) == TRUE) {
3635         reboot_flag = lm_get_char();
3636
3637         reboot_delay_time = lm_get_int() * 1000, /* ms */
3638
3639         time_reboot_was_issued = lm_time();
3640         rebooting = TRUE;
3641     }
3642     else {
3643         lm_queue_message(ERROR_MSG, "Password required to reboot modeler");
3644     }
3645     #else
3646     lm_queue_message(ERROR_MSG, "Core Modeler Code is not running on modeler; command ignored...");
3647     #endif
3648
3649     end_queue_message();
3650     end_put(user->id);
3651 }
3652
3653 void process_shutdown_cmd(user)
3654 USER_INFO *user;
3655 {
3656     DPRINTF(("inside process_shutdown_cmd\n"));
3657
3658     reset_obuf();
3659     lm_put_int(SHUTDOWN_ANS);
3660
3661     #ifdef MODELER
3662     if (check_password(user) == TRUE) {
3663         reboot_flag = lm_get_char();
3664
3665         reboot_delay_time = lm_get_int() * 1000, /* ms */
3666
3667         time_reboot_was_issued = lm_time();
3668         rebooting = TRUE;
3669         shutdown = TRUE;
3670     }
3671     else {
3672         lm_queue_message(ERROR_MSG, "Password required to shutdown modeler");
3673     }
3674     #else
3675     lm_queue_message(ERROR_MSG, "Core Modeler Code is not running on modeler; command ignored...");
3676     #endif
3677
3678     end_queue_message();
3679     end_put(user->id);
3680 }
3681
3682 void process_password_cmd(user)
3683 USER_INFO *user;
3684 {
3685     u_short attrib;
3686     char password[MAX_STRING_LENGTH];
3687     char crypt_password[CRYPTED_PASSWORD_LENGTH];
3688     short i;
3689
3690     DPRINTF(("inside process_password_cmd\n"));
3691
3692     reset_obuf();
3693     lm_put_int(PASSWORD_ANS);
3694     attrib = lm_get_int();
3695
3696     i = 0;
3697     while (password[i++] != lm_get_char())
3698         ;
3699
3700     lm_crypt(password, crypt_password);
3701
3702     switch (attrib) {
3703         case LM_ASSIGN:
3704             if (check_password(user) == TRUE) {
3705                 if (write_password(crypt_password) == SUCCESS) {
3706                     for (i = 0; i < MAX_USER_COUNT; ++i) {
3707                         user_info_array[i]->good_password = FALSE;
3708                     }
3709                     user->good_password = TRUE;
3710                 }
3711             }
3712             else {
3713                 lm_queue_message(ERROR_MSG, "Password required to set new password on modeler");
3714             }
3715             break;
3716         case LM_ENTER:
3717             user->good_password = compare_password(crypt_password);
3718     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/function.c		DATE 5/23/89	PAGE # 32/49
				TIME 6:14:38 pm	
LINE #	SOURCE TEXT				
3719	break;				
3720	default:				
3721	lm_queue_message(ERROR_MSG, "illegal attribute: %d",				
3722	attrib);				
3723	break;				
3724	}				
3725					
3726	end_queue_message();				
3727	end_put(user->fd);				
3728					
3729					
3730					

1421

5,353,243

1422

Copyright 1989  
Logic Modeling SystemsHEADER FILE  
lm1000/function.hDATE 5/23/89 PAGE #  
TIME 6:14:41 pm 1/50

LINE # HEADER TEXT

```
1 /* SCIS_ID: function.h rev 3.1, 4/24/89 at 07:52:59 */
2
3 extern void process_create_def_cmd();
4 extern void process_create_instance_cmd();
5 extern void process_create_fault_cmd();
6
7 extern void process_release_def_cmd();
8 extern void process_release_instance_cmd();
9 extern void process_release_fault_cmd();
10
11 extern void process_eval_cmd();
12
13 extern void process_save_def_cmd();
14 extern void process_save_def_cont_cmd();
15 extern void process_save_ptrn_cmd();
16 extern void process_save_ptrn_cont_cmd();
17 extern void process_restore_inst_cmd();
18 extern void process_restore_ptrn_cmd();
19 extern void process_ptrn_hint_cmd();
20
21 extern void process_inq_modeler_cmd();
22 extern void process_inq_user_list_cmd();
23 extern void process_inq_user_cmd();
24 extern void process_inq_lane_cmd();
25 extern void process_inq_pam_cmd();
26 extern void process_inq_pel_cmd();
27 extern void process_inq_device_list_cmd();
28 extern void process_inq_device_name_cmd();
29 extern void process_inq_device_cmd();
30 extern void process_inq_dab_cmd();
31 extern void process_inq_dab_loc_cmd();
32 extern void process_inq_instance_cmd();
33 extern void process_inq_fault_cmd();
34
35 extern void process_inq_avail_ptrn_cmd();
36
37 extern void process_tmeasuremnt_cmd();
38
39 extern void process_loop_ptrn_cmd();
40
41 extern void process_reset_inst_cmd();
42
43 extern void process_test_network_cmd();
44
45 extern void process_abort_cmd();
46
47 extern void process_begin_session_cmd();
48
49 extern void process_label_dab_cmd();
50
51 extern void process_eval_control_cmd();
52
53 extern void process_reboot_cmd();
54 extern void process_shutdown_cmd();
55
56 extern void process_check_dabdef_cmd();
57
58 extern void process_password_cmd();
59
60 extern void process_lm_read();
61 extern void process_lm_write();
62
63 extern void process_no_such();
```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hardware.c

DATE 5/23/89  
TIME 6:14:41 pm

PAGE #  
1/51

```

LINE # SOURCE TEXT
1  /* SCCS_ID: hardware.c rev 3.1, 4/24/89 at 07:53:02 */
2
3  #include "device.h"
4  #include "hardware.h"
5  #include "eeprom.h"
6  #include "lmservr.h"
7  #include "id.h"
8
9  #define LONG 0
10 #define SHORT 1
11 #define CHAR 2
12
13 extern void id_load();
14
15 typedef struct {
16     DAB_EEPROM dab_eeprom;
17     u_long active;
18 } DAB_EEPROM_INFO;
19
20 DAB_EEPROM_INFO dab_eeprom_info[MAX_LANE_COUNT][MAX_SLOT_COUNT];
21
22 /*-----*/
23
24 init_dab_eeprom_info()
25 {
26     u_char laneso;
27     u_char slotsso;
28
29     for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
30         for (slotsso = 0; slotsso < MAX_SLOT_COUNT; ++slotsso) {
31             dab_eeprom_info[laneso][slotsso].active = FALSE;
32         }
33     }
34 }
35
36 u_char
37 read_loc_char(address)
38 u_char *address;
39 {
40
41     #ifdef MODELER
42         return(*address);
43     #else
44     #ifdef DIRECTCONN
45         u_long value;
46
47         dc_read_loc(CHAR, (u_long)address, &value);
48
49         return(value & 0xff);
50     #else
51         u_long value;
52
53         (void)db_fetch((u_long)address, &value);
54
55         return(value);
56     #endif
57     #endif
58 }
59
60 /*-----*/
61
62 u_short
63 read_loc_short(address)
64 u_short *address;
65 {
66
67     #ifdef MODELER
68         return(*address);
69     #else
70     #ifdef DIRECTCONN
71         u_long value;
72
73         dc_read_loc(SHORT, (u_long)address, &value);
74
75         return(value & 0xffff);
76     #else
77         u_long value;
78
79         (void)db_fetch((u_long)address, &value);
80
81         return(value);
82     #endif
83     #endif
84 }
85
86 /*-----*/
87
88 u_long
89 read_loc_long(address)
90 u_long *address;
91 {
92     /*
93      * returns the contents of the location at address
94      */
95     #ifdef MODELER
96         return(*address);
97     #else
98     #ifdef DIRECTCONN
99         u_long value;
100
101         dc_read_loc(LONG, (u_long)address, &value);
102
103         return(value);
104     #else
105         u_long value;
106
107         (void)db_fetch((u_long)address, &value);
108
109         return(value);
110     #endif
111     #endif
112 }
113
114 /*-----*/
115
116 void
117 write_loc_char(address, value)
118 u_char *address;
119 u_char value;
120

```



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/hardware.c

DATE

5/23/89

PAGE #

TIME

6:14:41 pm

2/52

```

LINE # SOURCE TEXT
121 {
122 /*
123  * write value into location address
124  */
125
126 #ifdef MODELER
127     *address = value;
128 #else
129 #ifdef DIRECTCONN
130     dc_write_loc(CHAR, (u_long)address, value);
131 #else
132     (void)db_write((u_long)address, value);
133 #endif
134 #endif
135 }
136
137 void
138 write_loc_short(address, value)
139     u_short *address;
140     u_short value;
141 {
142     /*
143      * write value into location address
144      */
145
146 #ifdef MODELER
147     *address = value;
148 #else
149 #ifdef DIRECTCONN
150     dc_write_loc(SHORT, (u_long)address, value);
151 #else
152     (void)db_write((u_long)address, value);
153 #endif
154 #endif
155 }
156
157 void
158 write_loc_long(address, value)
159     u_long *address;
160     u_long value;
161 {
162     /*
163      * write value into location address
164      */
165
166 #ifdef MODELER
167     *address = value;
168 #else
169 #ifdef DIRECTCONN
170     dc_write_loc(LONG, (u_long)address, value);
171 #else
172     (void)db_write((u_long)address, value);
173 #endif
174 #endif
175 }
176
177 /*-----*/
178
179 probe(addr)
180     u_long addr;
181 {
182
183 #ifdef MODELER
184     return(lm_read_probe(addr));
185 #else
186 #ifdef DIRECTCONN
187     return(dc_probe(addr));
188 #else
189     u_long junk;
190
191     return(db_fetch(addr, &junk));
192 #endif
193 #endif
194 }
195
196 /*-----*/
197
198 read_cpu(tot_system_mem, slot_count, lane_count)
199     u_long *tot_system_mem;
200     u_char *slot_count;
201     u_char *lane_count;
202 {
203     /*
204      * return the tot_system_mem, and clab_board_id.
205      * This function better not fail !!
206      */
207     ID_PROM_CPU cpu_id_prom;
208     u_long status;
209     u_short model_number;
210     u_char dram_size;
211
212 #ifdef MODELER
213     id_load((u_char *)CPU_ID_PROM_ADDR, (char *)&cpu_id_prom);
214
215     dram_size = cpu_id_prom.dram_size;
216     *tot_system_mem = (dram_size + 1) * ID_CPU_DRAM_SIZE_K;
217
218     model_number = cpu_id_prom.model_number;
219
220     if (model_number == 1000) {
221         *slot_count = 8;
222         *lane_count = 4;
223     }
224
225     return(SUCCESS);
226 #else
227 #ifdef DIRECTCONN
228     *tot_system_mem = 4 * ID_CPU_DRAM_SIZE_K;
229
230     model_number = 1000;
231
232     if (model_number == 1000) {
233         *slot_count = 8;
234         *lane_count = 4;
235     }
236
237     return(SUCCESS);
238 #else
239     *tot_system_mem = read_loc_long((u_long *)CPU_MEM_SIZE_ADDR);
240

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/hardware.c	DATE 5/23/89	PAGE # 3/53
LINE #	SOURCE TEXT			
241	*slot_count = read_loc_long((u_long *)CPU_SLOT_COUNT_ADDR);			
242	*lane_count = read_loc_long((u_long *)CPU_LANE_COUNT_ADDR);			
243	return(SUCCESS);			
244	#endif			
245	#endif			
246	}			
247	}			
248	}			
249	/*-----*/			
250	read_pac(lane_number)			
251	u_char lane_number;			
252	{			
253	/*			
254	* return the pac_board_id and the slot_count of the lane_number.			
255	* return 0 if lane does not exist else return 1.			
256	*/			
257	u_long addr;			
258	addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +			
259	LANE_PAC_START_OFFSET + PAC_BOARD_ID_OFFSET;			
260	if (probe(addr) == FAILURE)			
261	return(FAILURE);			
262	return(SUCCESS);			
263	}			
264	}			
265	/*-----*/			
266	read_pel(lane_number, slot_number, pel_board_id)			
267	u_char lane_number;			
268	u_char slot_number;			
269	u_long *pel_board_id;			
270	{			
271	/*			
272	* return the board_id of the PEL on geographical			
273	* address lane_number and slot addr.			
274	* return 0 if PEL does not exist else return 1.			
275	*/			
276	u_long addr;			
277	/* check if PEL exists */			
278	addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +			
279	LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +			
280	PEL_ID_FROM_OFFSET;			
281	if (probe(addr) == FAILURE)			
282	return(FAILURE);			
283	*pel_board_id = read_loc_long((u_long *)addr);			
284	return(SUCCESS);			
285	}			
286	}			
287	/*-----*/			
288	read_tmg(tmg_board_id)			
289	u_long *tmg_board_id;			
290	{			
291	/*			
292	* return the board_id of the Timing Generator			
293	*/			
294	u_long addr;			
295	addr = CLOS_START_ADDR + CLOS_BOARD_ID_OFFSET;			
296	if (probe(addr) == FAILURE)			
297	return(FAILURE);			
298	*tmg_board_id = read_loc_long((u_long *)addr);			
299	return(SUCCESS);			
300	}			
301	/*-----*/			
302	check_dab_present(lane_number, slot_number)			
303	u_char lane_number;			
304	u_char slot_number;			
305	{			
306	u_long addr;			
307	u_short word;			
308	dab_eeprom_info(lane_number)(slot_number).active = FALSE;			
309	if (dab_present(lane_number, slot_number) == FAILURE) {			
310	return(FAILURE);			
311	}			
312	addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +			
313	LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +			
314	PEL_STATUS_CONTROL_OFFSET;			
315	word = read_loc_long((u_long *)addr);			
316	/* assert RESET with INITIALIZE 0 in anticipation for an			
317	* EEPROM write later.			
318	*/			
319	word  = PEL_CS_RESET_MASK & PEL_CS_INITIALIZE_MASK;			
320	write_loc_long((u_long *)addr, (u_long)word);			
321	return(SUCCESS);			
322	}			
323	}			
324	/*-----*/			
325	read_dab(lane_number, slot_number, man_id, dab_type, revision, part_name,			
326	model_maker, model_revision,			
327	dab_shape, dab_segment_number, insertion_count)			
328	u_char lane_number;			
329	u_char slot_number;			
330	char *man_id;			
331	char *dab_type;			
332	char *revision;			
333	char *part_name;			
334	char *model_maker;			
335	char *model_revision;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hardware.c

DATE 5/23/89

PAGE #

TIME 6:14:41 pm

4/54

```

LINE # SOURCE TEXT
361 u_long *dab_shape;
362 u_char *dab_segment_number;
363 u_short *insertion_count;
364 {
365     /*
366     * Return the info stored in the EEPROM for the DAB on
367     * geographical address lane_number and slot_addr.
368     * return 0 if DAB does not exist else return 1.
369     */
370
371 #ifdef DBASE
372     EEPROM_CONTENTS *eeprom;
373 #endif
374     DAB_EEPROM dab_eeprom;
375     u_long addr;
376     u_short word;
377     u_char i;
378
379     DPRINTF(("inside read_dab() lane: %d slot: %d\n",
380             lane_number, slot_number));
381
382     if (dab_present(lane_number, slot_number) == FAILURE) {
383         DPRINTF(("dab absent\n"));
384         dab_eeprom_info[lane_number][slot_number].active = FALSE;
385         return(FAILURE);
386     }
387
388     DPRINTF(("dab present\n"));
389
390 #ifdef DBASE
391     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
392            LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +
393            PEL_STATUS_CONTROL_OFFSET;
394
395     /* Turn off the 1s use LED */
396     word = read_loc_long((u_long *)addr) & 0xffff & PEL_CS_IN_USE_LED_MASK;
397     write_loc_long((u_long *)addr, (u_long)word);
398
399     DPRINTF(("rd: 1s: %d sl: %d pel status: %04X\n",
400             lane_number, slot_number, word));
401
402     /* If the ACTIVE bit is set then we already know the contents of EEPROM */
403     if (((word & PEL_CS_ACTIVE_MASK) != 0) &&
404         (dab_eeprom_info[lane_number][slot_number].active == TRUE)) {
405         dab_eeprom = dab_eeprom_info[lane_number][slot_number].dab_eeprom;
406     }
407     else {
408         DPRINTF(("calling lm_read_eeprom() lane: %d slot: %d\n",
409                 lane_number, slot_number));
410         /* assert RESET */
411         word = PEL_CS_RESET_MASK;
412         write_loc_long((u_long *)addr, (u_long)word);
413
414         if (lm_read_eeprom(lane_number * MAX_SLOT_COUNT + slot_number,
415                             &dab_eeprom) == FAILURE) {
416             init_magic_and_pel(lane_number, slot_number);
417             DPRINTF(("failure in lm_read_eeprom()\n"));
418             dab_eeprom_info[lane_number][slot_number].active = FALSE;
419             return(FAILURE);
420         }
421
422         dab_eeprom_info[lane_number][slot_number].active = TRUE;
423         dab_eeprom_info[lane_number][slot_number].dab_eeprom = dab_eeprom;
424     }
425
426     for (i = 0; i < NAME_LENGTH; ++i)
427         part_name[i] = dab_eeprom.device_name[i];
428     part_name[i] = '\0';
429
430     for (i = 0; i < MAKER_LENGTH; ++i)
431         model_maker[i] = dab_eeprom.model_maker[i];
432     model_maker[i] = '\0';
433
434     for (i = 0; i < REV_LENGTH; ++i)
435         model_revision[i] = dab_eeprom.model_revision[i];
436     model_revision[i] = '\0';
437
438     for (i = 0; i < MAN_LENGTH; ++i)
439         man_id[i] = dab_eeprom.dab_manufacturer[i];
440     man_id[i] = '\0';
441
442     for (i = 0; i < REV_LENGTH; ++i)
443         revision[i] = dab_eeprom.dab_revision[i];
444     revision[i] = '\0';
445
446     for (i = 0; i < TYPE_LENGTH; ++i)
447         dab_type[i] = dab_eeprom.dab_type[i];
448     dab_type[i] = '\0';
449
450     *dab_shape = dab_eeprom.configuration;
451     *dab_segment_number = dab_eeprom.segment_number;
452     *insertion_count = dab_eeprom.insertion_count;
453
454     if (*dab_shape == 0) {
455         init_magic_and_pel(lane_number, slot_number);
456         DPRINTF(("dab shape is 0\n"));
457         dab_eeprom_info[lane_number][slot_number].active = FALSE;
458         return(FAILURE);
459     }
460
461     return(SUCCESS);
462 }
463
464 #else
465     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
466            LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC;
467
468     if (!db_fetch_array(addr + PEL_DAB_EEPROM_OFFSET, (char **)&eeprom))
469         return(FAILURE);
470
471     (void)strcpy(man_id, eeprom->man_id);
472     (void)strcpy(dab_type, eeprom->dab_type);
473     (void)strcpy(revision, eeprom->revision);
474     (void)strcpy(part_name, eeprom->part_name);
475     *dab_shape = eeprom->dab_shape;
476     *dab_segment_number = eeprom->dab_segment_number;
477     *insertion_count = 0;
478     return(SUCCESS);
479 #endif
480 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hardware.c

DATE 5/23/89  
TIME 6:14:41 pm

PAGE #  
5/55

```

LINE # SOURCE TEXT
481
482 /* ???
483 if (! db_fetch_array(addr + 0xb00, (char **) &oeprcm))
484     return(FAILURE);
485
486 {void} strcpy(mas_id, oeprcm->mas_id);
487 {void} strcpy(dab_type, oeprcm->dab_type);
488 {void} strcpy(revname, oeprcm->revname);
489 {void} strcpy(part_name, oeprcm->part_name);
490 *dab_shape = oeprcm->dab_shape;
491 *dab_segment_number = oeprcm->dab_segment_number;
492 *insertion_count = 0;
493 return(SUCCESS);
494 }
495
496
497 dab_present(lane_number, slot_number)
498 u_char lane_number;
499 u_char slot_number;
500 {
501     u_long addr;
502     u_long status;
503
504     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
505           LANE_PEL_0_START_OFFSET + slot_number * LANE_PEL_ADDR_INC +
506           PEL_STATUS_CONTROL_OFFSET;
507
508     /* PEL does not exist */
509     if (system_config->lane(lane_number)->pel(slot_number) == NULL)
510         return(FAILURE);
511
512     status = read_loc_long((u_long *)addr);
513     DPRINTF(("dp: ls: %d sl: %d pel status: %08x\n",
514             lane_number, slot_number, status));
515
516     if (status & PEL_CS_PRESENT_MASK)
517         return(SUCCESS);
518
519     return(FAILURE);
520 }
521
522
523
524 read_pam_count_reg(lane_number, count)
525 u_char lane_number;
526 u_char *count;
527 {
528     u_long addr;
529     u_long temp;
530
531     *count = 0;
532
533     if (read_pac(lane_number) == FAILURE)
534         return(FAILURE);
535
536     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
537           LANE_PAC_START_OFFSET + PAC_PAM_COUNT_REG_OFFSET;
538
539     temp = read_loc_long((u_long *)addr) & 0xf;
540
541     switch (temp) {
542     case PAC_PAM_COUNT_EQ_0:
543         *count = 0;
544         break;
545     case PAC_PAM_COUNT_EQ_1:
546         *count = 1;
547         break;
548     case PAC_PAM_COUNT_EQ_2:
549         *count = 2;
550         break;
551     case PAC_PAM_COUNT_EQ_3:
552         *count = 3;
553         break;
554     case PAC_PAM_COUNT_EQ_4:
555         *count = 4;
556         break;
557     default:
558         *count = 0;
559         break;
560     }
561     return(SUCCESS);
562 }
563
564
565
566 read_pam_present(lane_number, panno, value)
567 u_char lane_number;
568 u_char panno;
569 u_char *value;
570 {
571     /* If PAM present --> return "value" = 1 otherwise return "value" = 0 */
572
573     u_long addr;
574
575     *value = 0;
576     if (read_pac(lane_number) == FAILURE)
577         return(FAILURE);
578
579     addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
580           LANE_PAC_START_OFFSET + PAC_PAM_0_PRESENT_REG_OFFSET +
581           panno * PAC_PAM_PRESENT_REG_ADDR_INC;
582
583     #ifdef DBASE
584     /* PAM is present if DO == 0 */
585     *value = (read_loc_long((u_long *)addr) & 1) == 0;
586     #else
587     *value = (read_loc_long((u_long *)addr) & 1) == 1;
588     #endif
589
590     return(SUCCESS);
591 }
592
593
594
595 read_pam_id(lane_number, panno, pam_type, pam_board_id)
596 u_char lane_number;
597 u_char panno;
598 u_char pam_type;
599 u_long *pam_board_id;
600 {

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/hardware.c

DATE

5/23/89

PAGE #

TIME

6:14:41 pm

6/56

```

LINE # SOURCE TEXT
601 ID PROM_PAM pam_id_prom;
602 u_long addr;
603
604 if (read_pac(lane_number) == FAILURE)
605     return(FAILURE);
606
607 /* The IDPROM on the PAM is located at the LSB of the longword boundary,
608    * therefore we need to add 3 to the address.
609    */
610 addr = LANE_0_START_ADDR + lane_number * LANE_ADDR_INC +
611        LANE_PAC_START_OFFSET + PAC_PAM_0_ID_SIZE_REG_OFFSET +
612        pamno * PAC_PAM_ID_SIZE_REG_ADDR_INC + 3;
613
614 #ifdef MODELER
615 id_load((u_char *)addr, (char *)pam_id_prom);
616
617 switch (pam_id_prom.generic.board_type) {
618 case ID_BT_PAM128K:
619     *pam_type = PAC_PAM_128K;
620     break;
621 case ID_BT_PAM512K:
622     *pam_type = PAC_PAM_512K;
623     break;
624 case ID_BT_PAM2M:
625     *pam_type = PAC_PAM_2M;
626     break;
627 default:
628     *pam_type = PAC_PAM_128K;
629     break;
630 }
631
632 *pam_board_id = 0;
633
634 #else
635 *pam_type = PAC_PAM_128K;
636 *pam_board_id = 0;
637 #endif
638
639 return(SUCCESS);
640 }
641
642 /*-----*/
643 cpu_tag_interrupt_status()
644 {
645     u_long status;
646
647     /* This bit is active LOW */
648     status = read_loc_long((u_long *)CPU_STATUS_REG_ADDR);
649
650     if (status & CPU_TAG_INTR_MASK)
651         return(FALSE);
652     return(TRUE);
653 }
654
655 /*-----*/
656
657

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hex.c

DATE 5/23/89  
TIME 6:14:41 pm

PAGE #  
1/57

```

1  // SCLS_ID: hex.c rev 3.1. 4/24/89 at 07:53:05  //
2  //
3  //
4  // MAIN.C
5  // This file contains the startup code for the tasks.
6  // Tasks and resources begin here
7  //
8  #include "common.h"
9  #include "cpu.h"
10 #include "task.h"
11 #include "lance.h"
12 #include "mod_err.h"
13 #include "sparam.h"
14 #include "lm_rd_wr.h"
15 //
16 //
17 // The globals
18 //
19 BOOT_STRUCT boot;
20 extern unsigned long timer_semaphore, rcv_lance_pkt_semaphore;
21 int play_semaphore, malloc_semaphore;
22 //
23 //
24 //
25 // The BSP creates the main task, and looks for the entry point
26 // named main.
27 //
28 //
29 u_long auto_start = 0;
30 main()
31 {
32     void housekeeping_task();
33     void outputRoutine();
34     void tx_task();
35     void receive_task();
36     extern void set_boot();
37     extern u_short lm_sparam_access();
38     void pattern_task();
39     u_short write_lance_car(), read_lance_car();
40     int err;
41     extern u_long modeler_inet;
42     extern
43     unsigned long error;
44     //
45     // This routine initializes the partition used in the system.
46     // The only usage is by malloc and friends.
47     //
48     create_malloc_partition(&end, (u_long)CPU_RAM_SIZE - (u_long)&end);
49     //
50     // The LANCE requires that the STOP bit of CSR0 be set
51     // before accessing CSR0, CSR2, or CSR3.
52     if (write_lance_car(SELECT_CSR0, STOP_ACTIVITY) == FAILURE)
53     {
54         outputRoutine("Error trying to set stop");
55     }
56     if (lm_sparam_access((char *) &boot, BOOT, sizeof BOOT, MEMORY_READ, &error) == FAILURE)
57     {
58         outputRoutine("Failed to read NParam");
59     }
60     modeler_inet = boot.modeler_internet_address;
61     //
62     // Initialize the fifos
63     //
64     fifo_initialize();
65     //
66     // Create semaphore for timer
67     // 1 = init_value
68     // 0 = priority order
69     //
70     timer_semaphore = sc_create(1, 0, &err);
71     if (err) outputRoutine("Error creating timer semaphore");
72     //
73     //
74     // The malloc semaphore controls access to the malloc and free
75     // functions. This avoids re-entrancy problems associated with
76     // a multi-tasking environment.
77     //
78     malloc_semaphore = sc_create(1, 0, &err);
79     if (err) outputRoutine("Error creating malloc semaphore");
80     //
81     //
82     // end of play interrupt semaphore
83     //
84     play_semaphore = sc_create(0, 1, &err);
85     if (err) sys_out("Error creating event flag for EOP interrupt.");
86     //
87     //
88     // create semaphore for lance rcv ISR
89     // 0 = init_value
90     // 0 = priority order
91     //
92     rcv_lance_pkt_semaphore = sc_create(0, 0, &err);
93     if (err) outputRoutine("Error creating rcv lance pkt semaphore");
94     //
95     if (get_lance_ready_to_go() != SUCCESS) {
96         outputRoutine("Error getting the LANCE ready to go.");
97         sc_tasuspend(0, 0, &err);
98     }
99     //
100    if (start_lance() != SUCCESS) {
101        outputRoutine("Error starting the LANCE.");
102        sc_tasuspend(0, 0, &err);
103    }
104    //
105    // spin of the tasks
106    //
107    sc_tcreate(housekeeping_task, HOUSEKEEPING_TASK_ID,
108              &error);
109    if (err) outputRoutine("Error creating housekeeping task.");
110    //
111    //
112    //
113    sc_tcreate(tx_task, TRANSMIT_TASK_ID, TRANSMIT_TASK_PRI, &err);
114    if (err) outputRoutine("Error creating transmit task.");
115    //
116    //
117    sc_tcreate(receive_task, RECEIVE_TASK_ID, RECEIVE_TASK_PRI, &err);
118    if (err) outputRoutine("Error creating receive task.");
119    //
120    //
121    sc_tcreate(pattern_task, PATTERN_TASK_ID, PATTERN_TASK_PRI, &err);
122    if (err) outputRoutine("Error creating pattern task.");

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hex.c

DATE	5/23/89	PAGE #
TIME	6:14:41 pm	2/58

LINE #	SOURCE TEXT
121	
122	/* End of main task, main task commits suicide*/
123	sc_delete( 0, 0, &arr);
124	
125	/* end main */
126	
127	
128	/*.....*/
129	/* sys_out */
130	/* Sends message to serial Channel A */
131	/*.....*/
132	
133	sys_out( buf )
134	char *buf,
135	{
136	while( *buf )
137	{
138	sc_putc( *buf++ );
139	}
140	/* end sys_out */

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
1/59

```

1  /* SCCS ID: hwconfig.c rev 1.1, 4/24/89 at 07:53:08 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "mod_err.h"
7  #include "espram.h"
8  #include "laser.h"
9  #include "laser.h"
10 #include "network.h"
11 #include "cpu.h"
12
13 #ifdef MODELER
14 #include "vtx.h"
15 #endif
16
17
18 extern LM_HARDWARE_ERROR modeler_error;
19 extern CONFIGURATION_ERRORS config_error;
20
21 DAS_INFO *bad_dab_list[MAX_LANE_COUNT * MAX_SLOT_COUNT];
22
23 #ifdef DEASE
24 /* Set this variable to FALSE (using adb) to skip TNG calibration. */
25 do_tng_calibration = TRUE;
26 #else
27 do_tng_calibration = FALSE;
28 #endif
29
30 #define LIGHT_FAULT_LED() \
31     (((cpu_control_reg_struct *)CPU_CONTROL_REG)->fault_led = LED_ON)
32
33 read_hw_config()
34 {
35     /* Read the hardware configuration and fill in the data structure */
36
37     SYSTEM_INFO *new_system_info();
38     LANE_INFO *new_lane_info();
39     PAM_INFO *new_pam_info();
40     PEL_INFO *new_pel_info();
41     LANE_INFO *lane_ptr;
42     PEL_INFO *pel_ptr;
43     u_long board_id;
44     u_char lanes0, panno, slotno;
45     u_char fb_block_index;
46
47 #ifdef DIRECTCONN
48     dc_init();
49 #endif
50
51     init_dab_espram_info();
52
53     system_config = new_system_info();
54
55     (void)read_cpu(&system_config->tot_system_mem,
56                  &system_config->slot_count,
57                  &system_config->phy_lane_count);
58
59     if (read_tng(&system_config->clab_board_id) == FAILURE) {
60         LIGHT_FAULT_LED();
61         lm_config_error(CERR_NO_TNG, 0, 0);
62         write_config_error();
63         return;
64     }
65
66 #ifdef DIRECTCONN
67     if (dc_init_pac() == FAILURE) {
68         DPRINTF(("Error from dc_init_pac\n"));
69         exit(1);
70     }
71 #else
72 #ifdef MODELER
73     if (backplane_reset() == FAILURE)
74         DPRINTF(("ERROR in backplane_reset()\n"));
75 #else
76     /* DEASE: don't do anything */
77 #endif
78 #endif
79
80     if (do_tng_calibration == TRUE) {
81         DPRINTF(("calling tng_calibrate()\n"));
82         if (tng_calibrate() == FAILURE) {
83             DPRINTF(("FAILURE in tng_calibrate()\n"));
84             LIGHT_FAULT_LED();
85             lm_config_error(CERR_TNG_CAL, 0, 0);
86             write_config_error();
87             return;
88         }
89     }
90
91     for (lanes0 = 0; lanes0 < MAX_LANE_COUNT; lanes0++) {
92         /* check if the lane exists */
93         lane_ptr = new_lane_info();
94         system_config->lane[lanes0] = lane_ptr;
95
96         if (read_pac(lanes0) == SUCCESS) {
97             DPRINTF(("found lane: %d\n", lanes0));
98             lane_ptr->pac_present = TRUE;
99
100             (void)configure_pam(lanes0, lane_ptr);
101
102             fb_block_index = 0;
103             for (panno = 0; panno < MAX_PAM_COUNT; panno++) {
104                 if (lane_ptr->pam[panno] != NULL) {
105                     if (fb_block_index == 0)
106                         fb_block_count_array[lanes0].PAM_max_addr[fb_block_index] =
107                             lane_offset(lanes0) +
108                             lane_ptr->pam[panno]->mem_size * PTRN_ADDR_INC;
109                     else
110                         fb_block_count_array[lanes0].PAM_max_addr[fb_block_index] =
111                             fb_block_count_array[lanes0].PAM_max_addr[fb_block_index - 1] +
112                             lane_ptr->pam[panno]->mem_size * PTRN_ADDR_INC;
113                     fb_block_count_array[lanes0].
114                         PAM_max_addr[fb_block_index] =
115                             lane_ptr->pam[panno]->mem_size * PTRN_ADDR_INC;
116                     fb_block_count_array[lanes0].
117                         feedback_block_count[fb_block_index++] =
118                             fb_block_count_for_PAM(lane_ptr->pam[panno]->mem_size);
119                 }
120             }
121         }
122     }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE	5/23/89	PAGE #
TIME	6:14:42 pm	2/60

```

121 }
122 }
123
124 /* find all of the PEL's in this lane */
125 for (slotno = 0; slotno < MAX_SLOT_COUNT; slotno++) {
126     if (read_pel(lane--, slotno, &board_id)) {
127         pel_ptr = new_pel_info();
128         lane_ptr->pel[slotno] = pel_ptr;
129         pel_ptr->board_id = board_id;
130     }
131 }
132 }
133
134 init_bad_dab_list();
135
136 read_dab_configuration(dab_list);
137
138 verify_dab_list(dab_list);
139
140 /* We have to reset MAGIC ERRORS for ALL magic chips in the system; the
141    * good ones and the bad ones.
142    */
143
144 reset_magic_error(dab_list);
145 reset_magic_error(bad_dab_list);
146
147 raise_pel_reset();
148
149 rls_bad_dab_list();
150
151 measure_out_vcc(dab_list);
152
153 build_free_block_list();
154
155 write_config_error();
156 }
157
158 build_free_block_list()
159 {
160     LANE_INFO *lane_ptr;
161     u_long total_mem;
162     u_long link_table_addr;
163     u_long max_link_table_addr;
164     u_long block_addr;
165     u_short i;
166     u_short block_count;
167     u_char lane_no;
168     u_char panno;
169
170     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
171         lane_ptr = system_config->lane[lane_no];
172
173         if (lane_ptr->pac_present == FALSE) {
174             free_block_list[lane_no] = NULL;
175             continue;
176         }
177
178         /* Calculate total pattern memory */
179         total_mem = 0;
180         for (panno = 0; panno < MAX_PAN_COUNT; ++panno) {
181             if (lane_ptr->pan[panno] != NULL) {
182                 total_mem += lane_ptr->pan[panno]->mem_size;
183             }
184         }
185
186         if (total_mem) {
187
188             /* make the free_block_list point to block number 1 (second block) */
189             free_block_list[lane_no] = lane_offset(lane_no) +
190                 BLOCK_ADDR_INC;
191
192             /* Mark all blocks as being free in the link table */
193             link_table_addr = lane_offset(lane_no) + LANE_LINK_TABLE_OFFSET;
194
195             block_count = total_mem / PTEN_PER_BLOCK;
196             max_link_table_addr = link_table_addr +
197                 block_count * LINK_TABLE_ADDR_INC;
198
199             while (link_table_addr != max_link_table_addr) {
200                 write_loc_long((u_long *)link_table_addr, (u_long)FREE_BLOCK_FLAG);
201                 link_table_addr += LINK_TABLE_ADDR_INC;
202             }
203
204             /* link all of the blocks */
205             block_addr = free_block_list[lane_no];
206             for (i = 1; i < block_count; ++i) {
207                 /* write forward link */
208                 write_loc_long((u_long *)block_addr, block_addr + BLOCK_ADDR_INC);
209
210                 /* write backward link */
211                 write_loc_long((u_long *)block_addr + 4,
212                     block_addr - BLOCK_ADDR_INC);
213
214                 block_addr += BLOCK_ADDR_INC;
215             }
216
217             /* Fix the backward pointer of the HEAD of the list */
218             write_loc_long((u_long *)free_block_list[lane_no] + 4, (u_long)NULL);
219
220             /* Fix the forward pointer of the TAIL of the list */
221             write_loc_long((u_long *)block_addr - BLOCK_ADDR_INC, (u_long)NULL);
222         }
223         else
224             free_block_list[lane_no] = NULL;
225     }
226 }
227
228 long calculate_pel_count(lane_number)
229 u_char lane_number;
230 {
231     DAB_INFO *temp;
232     u_char index;
233     char i;
234     u_char total_pel = 0;
235     u_char dabno;
236     u_char calculate_pel_count_in_segment();
237 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
3/61

```

LINE # SOURCE TEXT
241 for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
242     if (dab_list[dabno] == NULL)
243         continue;
244     temp = dab_list[dabno];
245     if (temp->segment[0] != NULL)
246         total_pel += calculate_pel_count_in_segment(lane_number,
247             temp->segment[0]);
248     else {
249         index = 1;
250         for (i = temp->segment_count; i > 0; --i) {
251             total_pel += calculate_pel_count_in_segment(lane_number,
252                 temp->segment[index++]);
253         }
254     }
255     return(total_pel);
256 }
257
258 u_char calculate_pel_count_in_segment(lane_number, segment)
259 u_char lane_number;
260 SEGMENT_EL segment;
261 {
262     u_long mask;
263     u_long shape_bit_map;
264     u_char i;
265     u_char count = 0;
266     if (segment->lane_no > lane_number)
267         return(0);
268     else {
269         shape_bit_map = segment->shape_bit_map;
270         mask = 1 << (lane_number - segment->lane_no) * 8;
271         for (i = 0; i < 8; ++i, mask <<= 1) {
272             if (shape_bit_map & mask)
273                 ++count;
274         }
275     }
276     return(count);
277 }
278
279 dab_loc(dab_ptr, dab_no)
280 DAB_INFO *dab_ptr;
281 u_long dab_no;
282 {
283     u_long pro_number_of_locations_mark;
284     long *pro_lane_slot_array;
285     u_long shape_bit_map;
286     u_long mask;
287     u_char lane_no, slot_no;
288     u_char bit_no;
289     u_char seg_no;
290     u_char i, j;
291     u_char temp_lane;
292     u_char temp_slot;
293     u_char loc_count;
294     pro_number_of_locations_mark = LM_MARK_BUFFER(lm_global_cons_ptr);
295     lm_put_int(0); /* advance the obuf ptr */
296     pro_lane_slot_array = (long *)lm_get_cur_obuf_addr();
297     loc_count = 0;
298     if (dab_ptr->segment[0] != NULL) {
299         lane_no = dab_ptr->segment[0]->lane_no;
300         slot_no = dab_ptr->segment[0]->slot_no;
301         shape_bit_map = dab_ptr->segment[0]->shape_bit_map;
302         for (mask = 1, bit_no = 0; bit_no < 32; mask <<= 1, ++bit_no) {
303             if (mask & shape_bit_map) {
304                 lm_put_int((long)(lane_no + bit_no / 8));
305                 lm_put_int((long)(slot_no + bit_no % 8));
306                 ++loc_count;
307             }
308         }
309     }
310     else {
311         seg_no = dab_ptr->segment[0];
312         lane_no = dab_ptr->segment[seg_no]->lane_no;
313         slot_no = dab_ptr->segment[seg_no]->slot_no;
314         shape_bit_map = dab_ptr->segment[seg_no]->shape_bit_map;
315         for (mask = 1, bit_no = 0; bit_no < 32; mask <<= 1, ++bit_no) {
316             if (mask & shape_bit_map) {
317                 lm_put_int((long)(lane_no + bit_no / 8));
318                 lm_put_int((long)(slot_no + bit_no % 8));
319                 ++loc_count;
320             }
321         }
322     }
323     LM_PUT_LONG_AT_MARK(pro_number_of_locations_mark, lm_global_cons_ptr,
324         loc_count);
325     /* Now sort the array according to the lane number */
326     for (i = 0; i < loc_count - 1; ++i) {
327         for (j = i + 1; j < loc_count; ++j) {
328             if (pro_lane_slot_array[i*2] > pro_lane_slot_array[j*2]) {
329                 temp_lane = pro_lane_slot_array[i*2];
330                 temp_slot = pro_lane_slot_array[i*2+1];
331                 pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2];
332                 pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1];
333                 pro_lane_slot_array[j*2] = temp_lane;
334                 pro_lane_slot_array[j*2+1] = temp_slot;
335             }
336         }
337     }
338     /* Then sort the array according to the slot number */
339     for (i = 0; i < loc_count - 1; ++i) {
340         for (j = i + 1; j < loc_count; ++j) {
341             if (pro_lane_slot_array[i*2] == pro_lane_slot_array[j*2]) {
342                 if (pro_lane_slot_array[i*2+1] > pro_lane_slot_array[j*2+1]) {
343                     temp_lane = pro_lane_slot_array[i*2];
344                     temp_slot = pro_lane_slot_array[i*2+1];
345                     pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2];
346                     pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1];
347                     pro_lane_slot_array[j*2] = temp_lane;
348                     pro_lane_slot_array[j*2+1] = temp_slot;
349                 }
350             }
351         }
352     }
353 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89 PAGE #  
TIME 6:14:42 pm 4/62

```

LINE # SOURCE TEXT
361     pro_lane_slot_array[i*2] = pro_lane_slot_array[j*2],
362     pro_lane_slot_array[i*2+1] = pro_lane_slot_array[j*2+1],
363
364     pro_lane_slot_array[j*2] = temp_lane;
365     pro_lane_slot_array[j*2+1] = temp_slot;
366
367 }
368
369 }
370
371
372 read_dab_configuration(loc_dab_list)
373 DAB_INFO *loc_dab_list[];
374 {
375     DAB_INFO *dab_ptr;
376     DAB_INFO *dab_ptr2;
377     SEGMENT_EL *seg;
378     DAB_INFO *new_dab_info();
379     SEGMENT_EL *new_segment();
380     u_long mask;
381     u_long dab_shape;
382     u_char ptr, *end;
383     u_short insertion_count;
384     u_char count;
385     dab_flag[MAX_LANE_COUNT][MAX_SLOT_COUNT];
386     char revision[REV_LENGTH + 1];
387     char dab_type[TYPE_LENGTH + 1];
388     char mas_id[MAN_LENGTH + 1];
389     char model_maker[MAKER_LENGTH + 1];
390     char model_revision[REV_LENGTH + 1];
391     u_char slotno;
392     u_char laneso;
393     char dab_info_index;
394     u_char dab_segment_sum;
395
396     ptr = (u_char *)dab_flag[0][0];
397     end = &dab_flag[MAX_LANE_COUNT][0]; /* points to loc just past the array */
398     while (ptr < end)
399         ptr += 0;
400
401     for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno)
402         loc_dab_list[slotno] = NULL;
403
404     for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
405         /* Find all of the DAB's in this lane */
406         dab_ptr = new_dab_info();
407         for (slotno = 0; slotno < MAX_SLOT_COUNT; slotno++) {
408             if (!dab_flag[laneso][slotno]) {
409                 dab_flag[laneso][slotno] = 1;
410
411                 if (read_dab(laneso, slotno,
412                     mas_id,
413                     dab_type,
414                     revision,
415                     dab_ptr->part_name,
416                     model_maker,
417                     model_revision,
418                     &dab_shape,
419                     &dab_segment_sum,
420                     &insertion_count) == SUCCESS) {
421
422                     for (count = 0; mask = 1; count < 32; mask <<= 1, ++count) {
423                         if (dab_shape & mask)
424                             dab_flag[laneso + count / 8][slotno + count % 8] = 1;
425                     }
426
427                     if ((short)(dab_info_index = find_dab(loc_dab_list,
428                         dab_ptr->part_name,
429                         (u_char)PUBLIC)) != -1)
430                         dab_ptr2 = loc_dab_list[dab_info_index];
431                     else
432                         dab_ptr2 = NULL;
433
434                     if (dab_ptr2 == NULL) {
435                         enter_dab_info(loc_dab_list, dab_ptr, laneso, slotno);
436
437                         seg = new_segment();
438
439                         dab_ptr->segment[dab_segment_sum] = seg;
440
441                         seg->shape_bit_map = dab_shape;
442                         seg->lane_no = laneso;
443                         seg->slot_no = slotno;
444                         seg->insertion_count = insertion_count;
445                         (void)strcpy(seg->revision, revision);
446                         (void)strcpy(seg->dab_type, dab_type);
447                         (void)strcpy(seg->mas_id, mas_id);
448                         (void)strcpy(seg->model_maker, model_maker);
449                         (void)strcpy(seg->model_revision, model_revision);
450
451                         dab_ptr = new_dab_info();
452                     }
453                 }
454             }
455             else {
456                 if (dab_segment_sum != 0) {
457                     if (dab_ptr2->segment[dab_segment_sum] != NULL) {
458                         /* The same kind of DAB has been seen before */
459                         DPRINTF(("error: duplicate segment\n"));
460                         lm_config_error(CERR_DUPLICATE_SEGMENT,
461                             laneso, slotno);
462                     }
463                     else {
464                         /* Other segment of the same part has been seen
465                          * before.
466                          */
467                         seg = new_segment();
468
469                         dab_ptr2->segment[dab_segment_sum] = seg;
470
471                         seg->shape_bit_map = dab_shape;
472                         seg->lane_no = laneso;
473                         seg->slot_no = slotno;
474                         seg->insertion_count = insertion_count;
475                         (void)strcpy(seg->revision, revision);
476                         (void)strcpy(seg->dab_type, dab_type);
477                         (void)strcpy(seg->mas_id, mas_id);
478                         (void)strcpy(seg->model_maker, model_maker);
479                         (void)strcpy(seg->model_revision, model_revision);
480

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
5/63

```

LINE #      SOURCE TEXT
481
482
483     }
484     enter_dab_info(loc_dab_list, dab_ptr, laneso, slotno);
485
486     seg = new_segment();
487
488     dab_ptr->segment[0] = seg;
489
490     seg->shape_bit_map = dab_shape;
491     seg->lane_no = laneso;
492     seg->slot_no = slotno;
493     seg->insertion_count = insertion_count;
494     (void)strcpy(seg->revision, revision);
495     (void)strcpy(seg->dab_type, dab_type);
496     (void)strcpy(seg->mag_id, mag_id);
497     (void)strcpy(seg->model_maker, model_maker);
498     (void)strcpy(seg->model_revision, model_revision);
499
500     dab_ptr = new_dab_info();
501
502     }
503
504     }
505
506     }
507     rls_dab_info(dab_ptr);
508
509 }
510
511 reset_magic_error(loc_dab_list)
512 DAB_INFO *loc_dab_list;
513 {
514     DAB_INFO *dab_ptr;
515     u_long dab_shape;
516     u_long junk;
517     u_long mask;
518     u_long magic_addr;
519     u_char lanesobase;
520     u_char slotsobase;
521     u_char laneso;
522     u_char slotno;
523     u_char dabno;
524     u_char count;
525     u_char i;
526
527     /* Reset each MAGIC chip error register in the system and also set the
528      * ACTIVE bit.
529      * Note that at this point the loc_dab_list has not been verified yet,
530      * so there could be missing segments.
531      */
532     for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
533         if (loc_dab_list[dabno] == NULL)
534             continue;
535
536         dab_ptr = loc_dab_list[dabno];
537
538         /* We don't need to reset magic if all units of the DABs are active -/
539         if (all_dab_unit_active_and_initialized(dab_ptr) == TRUE) {
540             DPRINTF(("dab: %d active --> don't reset\n", dabno));
541             continue;
542         }
543
544         DPRINTF(("reset dab: %d\n", dabno));
545
546         if (dab_ptr->segment[0] != NULL) {
547             /* This is a single segment device -/
548             lanesobase = dab_ptr->segment[0]->lane_no;
549             slotsobase = dab_ptr->segment[0]->slot_no;
550             dab_shape = dab_ptr->segment[0]->shape_bit_map;
551
552             for (count = 0, mask = 1; count < 32; mask <= 1, ++count) {
553                 if (dab_shape & mask) {
554                     laneso = lanesobase + count / 8;
555                     slotno = slotsobase + count % 8;
556
557                     if (read_pel(laneso, slotno, &junk) == SUCCESS) {
558                         /* We only need to read reg 15 from ONE of the MAGIC Chip
559                         * to reset the error register of ALL MAGIC Chips.
560                         */
561                         magic_addr = LANE_0_START_ADDR + laneso * LANE_ADDR_INC +
562                                     LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
563                                     PEL_MC_0_OFFSET;
564
565                         (void)read_loc_long((u_long *)
566                                              (magic_addr + MC_RESET_REG_OFFSET));
567
568                         /* Need to read some other Magic register to affect reset -/
569                         (void)read_loc_long((u_long *)
570                                              (magic_addr + MC_DATA_OUT_REG_OFFSET));
571
572                         init_dab(laneso, slotno);
573                     }
574                 }
575             }
576         }
577
578         /* This is a multi segments device -/
579         for (i = 0; i <= MAX_SEGMENT_PER_DAB_COUNT; ++i) {
580             if (dab_ptr->segment[i] == NULL)
581                 continue;
582
583             lanesobase = dab_ptr->segment[i]->lane_no;
584             slotsobase = dab_ptr->segment[i]->slot_no;
585             dab_shape = dab_ptr->segment[i]->shape_bit_map;
586
587             for (count = 0, mask = 1; count < 32; mask <= 1, ++count) {
588                 if (dab_shape & mask) {
589                     laneso = lanesobase + count / 8;
590                     slotno = slotsobase + count % 8;
591
592                     if (read_pel(laneso, slotno, &junk) == SUCCESS) {
593                         /* We only need to read reg 15 from ONE of the MAGIC Chip
594                         * to reset the error register of ALL MAGIC Chips.
595                         */
596                         magic_addr = LANE_0_START_ADDR + laneso * LANE_ADDR_INC +
597                                     LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
598                                     PEL_MC_0_OFFSET;
599                     }
600                 }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/hwconfig.c

DATE

5/23/89

PAGE #

TIME

6:14:42 pm

6/64

```

LINE # SOURCE TEXT
601
602 (void)read_loc_long((u_long *)
603 (magic_addr + MC_RESET_REG_OFFSET));
604
605 /* Need to read some other Magic reg to affect reset */
606 (void)read_loc_long((u_long *)
607 (magic_addr + MC_DATA_OUT_REG_OFFSET));
608
609 (void)read_loc_long((u_long *)magic_addr);
610
611 init_dab(laneno, slotno);
612
613 }
614
615 }
616
617 }
618
619
620 verify_dab_list(loc_dab_list);
621 DAB_INFO *loc_dab_list[];
622 {
623 /* Verify that
624 * - all of the PEL's exist to support the DAB's.
625 * - the PMA's exist in each lane occupied by the DAB
626 * - all segments of parts exists.
627 * - Duplicate devices have the same unit_location
628 * Set up the unit_addr field in the dab_info.
629 * Set up the last_in_lane field of the unit_addr_el
630 * Set up lane_wand[]
631 * Set up lane_count
632 * Set up unit_count_per_lane
633 * Set up the dummy_ptr
634 * Set up ident_lane
635 * Make the lane point to the most significant DAB.
636 */
637
638 DAB_INFO *dab_ptr;
639 DAB_INFO *dab_ptr2;
640 u_long mask;
641 u_long dab_shape;
642 u_char pel_location[MAX_LANE_COUNT][MAX_SLOT_COUNT];
643 u_char cur_unit;
644 u_char laneno;
645 u_char slotno;
646 u_char count;
647 u_char i, j;
648 u_char last_segment_num;
649 u_char dabno;
650 u_char dabno2;
651
652 for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {
653 if (loc_dab_list[dabno] == NULL)
654 continue;
655
656 dab_ptr = loc_dab_list[dabno];
657
658 if (dab_ptr->segment[0] != NULL) {
659 /* This is a single segment device */
660
661 dab_ptr->segment_count = 1;
662 cur_unit = 0;
663 laneno = dab_ptr->segment[0]->lane_no;
664 slotno = dab_ptr->segment[0]->slot_no;
665 dab_shape = dab_ptr->segment[0]->shape_bit_map;
666
667 for (count = 0; mask = 1; count < 32; mask <= 1, ++count) {
668 if (dab_shape & mask) {
669 dab_ptr->unit_location[cur_unit].lane_no = laneno + count / 8;
670 dab_ptr->unit_location[cur_unit].slot_no = slotno + count % 8;
671 ++cur_unit;
672 }
673 }
674
675 dab_ptr->unit_count = cur_unit;
676
677 if (check_pel_and_pam(dab_ptr) == FAILURE)
678 remove_dab_info(loc_dab_list, dabno);
679 else
680 set_last_in_lane(dab_ptr);
681 }
682 else {
683 /* This is a multi segments device */
684 /* Find the last segment */
685 for (i = MAX_SEGMENT_PER_DEVICE; i > 0; --i) {
686 if (dab_ptr->segment[i] != NULL)
687 break;
688 }
689
690 dab_ptr->segment_count = i;
691 cur_unit = 0;
692 last_segment_num = i;
693 for (i = 1; i <= last_segment_num; i++) {
694 if (dab_ptr->segment[i] == NULL)
695 break;
696 else {
697 laneno = dab_ptr->segment[i]->lane_no;
698 slotno = dab_ptr->segment[i]->slot_no;
699 dab_shape = dab_ptr->segment[i]->shape_bit_map;
700
701 for (count = 0; mask = 1; count < 32; mask <= 1, ++count) {
702 if (dab_shape & mask) {
703 if (cur_unit == MAX_UNIT_COUNT) {
704 for (j = 1; j <= last_segment_num; j++) {
705 lm_config_error(CERR_DEVICE_TOO_LARGE,
706 dab_ptr->segment[j]->lane_no,
707 dab_ptr->segment[j]->slot_no);
708 }
709
710 remove_dab_info(loc_dab_list, dabno);
711
712 goto next;
713 }
714
715 dab_ptr->unit_location[cur_unit].lane_no =
716 laneno + count / 8;
717 dab_ptr->unit_location[cur_unit].slot_no =
718 slotno + count % 8;
719
720

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/hwconfig.c	DATE 5/23/89	PAGE # 7/65
LINE #	SOURCE TEXT			
721	++cur_unit;			
722	}			
723	}			
724	}			
725	}			
726	}			
727	dab_ptr->unit_count = cur_unit;			
728	if (i != last_segment_num) {			
729	/* Not all of the segments of the device were found.			
730	* Consider this device unusable and remove it from the dab_list			
731	*/			
732	for (i = 1; i <= MAX_SEGMENT_PER_DEVICE; ++i) {			
733	if (dab_ptr->segment[i] == NULL)			
734	continue;			
735	lm_config_error(CERR_MISSING_SEGMENT,			
736	dab_ptr->segment[i]->lane_no,			
737	dab_ptr->segment[i]->slot_no);			
738	}			
739	}			
740	remove_dab_info(loc_dab_list, dabno);			
741	}			
742	else {			
743	/* All of the segments were found */			
744	if (check_pel_and_pem(dab_ptr) == FAILURE) {			
745	remove_dab_info(loc_dab_list, dabno);			
746	}			
747	else {			
748	/* Make the lane point to the most significant segment */			
749	set_last_lane(dab_ptr);			
750	}			
751	}			
752	}			
753	}			
754	}			
755	next:			
756	}			
757	}			
758	}			
759	/* Make sure that duplicate devices have the same relative unit location.			
760	* Note: only single segment devices can have duplicates at this point.			
761	*/			
762	for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT) - 1; ++dabno) {			
763	if (loc_dab_list[dabno] == NULL)			
764	continue;			
765	dab_ptr = loc_dab_list[dabno];			
766	for (dabno2 = dabno + 1;			
767	dabno2 < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno2) {			
768	if (loc_dab_list[dabno2] == NULL)			
769	continue;			
770	dab_ptr2 = loc_dab_list[dabno2];			
771	if (strcmp(dab_ptr->part_name, dab_ptr2->part_name) == 0) {			
772	if (cmp_dab_shape(dab_ptr, dab_ptr2) == FALSE) {			
773	lm_config_error(CERR_ILLEGAL_DUPLICATE_DEVICE,			
774	dab_ptr2->segment[0]->lane_no,			
775	dab_ptr2->segment[0]->slot_no);			
776	remove_dab_info(loc_dab_list, dabno2);			
777	}			
778	}			
779	/* Create a MAP of the PEL location to be used to check pel stacking */			
780	for (laneo = 0; laneo < MAX_LANE_COUNT; ++laneo) {			
781	for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) {			
782	pel_location[laneo][slotno] = 0;			
783	if (system_config->lane[laneo]->pel[slotno] != NULL)			
784	pel_location[laneo][slotno] = 1;			
785	}			
786	}			
787	}			
788	for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {			
789	dab_ptr = loc_dab_list[dabno];			
790	if (dab_ptr == NULL)			
791	continue;			
792	if (check_pel_stacking(dab_ptr, pel_location) == FAILURE) {			
793	if (dab_ptr->segment[0] != NULL) {			
794	/* This is a single segment device */			
795	lm_config_error(CERR_ILLEGAL_PEL_STACKING,			
796	dab_ptr->segment[0]->lane_no,			
797	dab_ptr->segment[0]->slot_no);			
798	remove_dab_info(loc_dab_list, dabno);			
799	}			
800	else {			
801	/* This is a multi segment device */			
802	for (i = 1; i <= MAX_SEGMENT_PER_DEVICE; ++i) {			
803	if (dab_ptr->segment[i] == NULL)			
804	continue;			
805	lm_config_error(CERR_ILLEGAL_PEL_STACKING,			
806	dab_ptr->segment[i]->lane_no,			
807	dab_ptr->segment[i]->slot_no);			
808	}			
809	remove_dab_info(loc_dab_list, dabno);			
810	}			
811	}			
812	}			
813	}			
814	}			
815	}			
816	}			
817	}			
818	}			
819	}			
820	}			
821	}			
822	}			
823	}			
824	}			
825	}			
826	}			
827	}			
828	}			
829	#ifdef DEBUG			
830	for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {			
831	if (loc_dab_list[dabno] == NULL)			
832	continue;			
833	dab_ptr = loc_dab_list[dabno];			
834	DPRINTF(("device name : %s", dab_ptr->part_name));			
835	DPRINTF(("unit_count_per_lane : %d", dab_ptr->unit_count_per_lane));			
836	DPRINTF(("segment_count : %d", dab_ptr->segment_count));			
837	DPRINTF(("lane_count : %d", dab_ptr->lane_count));			
838	DPRINTF(("unit_count : %d", dab_ptr->unit_count));			
839	}			
840	#endif			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
8/66

```

LINE # SOURCE TEXT
841 DPRINTF(("unit location : %d"),
842 for (i = 0; i < dab_ptr->unit_count; ++i) {
843     DPRINTF((" lane: %d slot: %d\n",
844         dab_ptr->unit_location[i].lane_no,
845         dab_ptr->unit_location[i].slot_no));
846 }
847 }
848 }
849 }
850 }
851 check_pel_and_pam(dab_ptr)
852 DAB_INFO *dab_ptr;
853 {
854     /* Check to make sure that each unit of the device is supported by a PEL,
855     * also that it is backed by PAM.
856     */
857     short i;
858     u_char laneso, slotso;
859     u_char any_error = FALSE;
860     for (i = dab_ptr->unit_count - 1; i >= 0; --i) {
861         laneso = dab_ptr->unit_location[i].lane_no;
862         slotso = dab_ptr->unit_location[i].slot_no;
863         /* Check if the PEL exists on this slot */
864         if (system_config->lane[laneso]->pel[slotso] == NULL) {
865             lm_config_error(CERR_NO_PEL_FOR_DAB,
866                 dab_ptr->unit_location[i].lane_no,
867                 dab_ptr->unit_location[i].slot_no);
868             any_error = TRUE;
869         }
870         /* Check if the PAC exists */
871         if (system_config->lane[laneso]->pac_present == FALSE) {
872             lm_config_error(CERR_NO_PAM_FOR_DAB,
873                 dab_ptr->unit_location[i].lane_no,
874                 dab_ptr->unit_location[i].slot_no);
875             any_error = TRUE;
876             continue;
877         }
878         /* Check if the PAM exists on this lane */
879         if (system_config->lane[laneso]->pam[0] == NULL) {
880             lm_config_error(CERR_NO_PAM_FOR_DAB,
881                 dab_ptr->unit_location[i].lane_no,
882                 dab_ptr->unit_location[i].slot_no);
883             any_error = TRUE;
884         }
885     }
886     if (any_error == FALSE)
887         return(SUCCESS);
888     else
889         return(FAILURE);
890 }
891
892 remove_dab_info(loc_dab_list, dabso)
893 DAB_INFO *loc_dab_list[];
894 u_char dabso;
895 {
896     /* Remove the dab_info from the loc_dab_list and put it to bad_dab_list. */
897     DAB_INFO *dab_ptr;
898     dab_ptr = loc_dab_list[dabso];
899     bad_dab_list[dabso] = dab_ptr;
900     loc_dab_list[dabso] = NULL;
901 }
902
903 init_dab(laneso, slotso)
904 u_char laneso;
905 u_char slotso;
906 {
907     /* Initialize the DAB (set the ACTIVE bit) at the laneso/slotso. */
908     u_long addr;
909     u_short word;
910     addr = LANE_0_START_ADDR + laneso * LANE_ADDR_INC +
911         LANE_PEL_0_START_OFFSET + slotso * LANE_PEL_ADDR_INC +
912         PEL_STATUS_CONTROL_OFFSET;
913     /* assert RESET */
914     word = read_loc_long((u_long *)addr);
915     DPRINTF(("id: %d al: %d pel status: %04x\n", laneso, slotso, word));
916     word |= PEL_CS_RESET_MASK;
917     write_loc_long((u_long *)addr, (u_long)word);
918     /* write INITIALIZE to 0 with RESET == 0 */
919     word = read_loc_long((u_long *)addr);
920     word = (word & PEL_CS_INITIALIZE_MASK);
921     write_loc_long((u_long *)addr, (u_long)word);
922     /* write INITIALIZE to 1 with RESET == 0 */
923     word = read_loc_long((u_long *)addr);
924     word = word | PEL_CS_INITIALIZE_MASK;
925     write_loc_long((u_long *)addr, (u_long)word);
926     /* deassert RESET */
927     word = read_loc_long((u_long *)addr);
928     word |= PEL_CS_RESET_MASK;
929     /* ??? disable magic error */
930     word |= PEL_CS_MAGIC_ERROR_ENABLE_MASK;
931     write_loc_long((u_long *)addr, (u_long)word);
932 }
933
934 reconfigure_dab()
935 {
936     DAB_INFO *dev_dab_list[MAX_LANE_COUNT*MAX_SLOT_COUNT];
937     EXTRA_DEVICE_SPEC *extra_def_ptr;
938     USER_INFO *user;
939     DEVICE_SPEC *definition;
940     INSTANCE_INFO *instance;
941     DAB_INFO *dab_ptr;
942     DAB_INFO *new_dab_ptr;
943     u_short dab_info_index;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
9/67

```

LINE # SOURCE TEXT
961 u_short      inst_id;
962 u_char      slotno;
963 u_char      userno;
964 u_char      again_count = 0;
965
966 again:
967 modeler_error.dab_change = FALSE;
968
969 DPRINTF(("waiting for DAB to settle\n"));
970 lm_delay((u_long)(1 - 1000));
971
972 if (modeler_error.error == TRUE) {
973     modeler_error.error = FALSE;
974
975     if ((modeler_error.pac_lane_errors != 0) ||
976         (modeler_error.tug_error == TRUE) ||
977         (modeler_error.unknown_source_of_interrupt == TRUE)) {
978         fatal_hardware_error_encountered = TRUE;
979         return;
980     }
981 }
982
983 /* We are going to reconfigure the DABs --> reinitialize the structure */
984 config_error.duplicate_segment = 0;
985 config_error.missing_segment = 0;
986 config_error.no_pal_for_dab = 0;
987 config_error.no_pam_for_dab = 0;
988 config_error.illegal_duplicate_device = 0;
989 config_error.device_too_large = 0;
990 config_error.illegal_pal_stacking = 0;
991 non_fatal_configuration_error_encountered = FALSE;
992
993 init_bad_dab_list();
994
995 read_dab_configuration(new_dab_list);
996
997 update_svars_dab_insertion(new_dab_list);
998
999 verify_dab_list(new_dab_list);
1000
1001 for (userno = 0; userno < MAX_USER_COUNT; ++userno) {
1002     user = user_info_array[userno];
1003
1004     if (user->active == FALSE)
1005         continue;
1006
1007     for (inst_id = 0; inst_id < user->inst_table_size; ++inst_id) {
1008         instance = user->instance[inst_id];
1009
1010         if (BOCUS_INSTANCE(user, instance))
1011             continue;
1012
1013         definition = instance->definition;
1014
1015         extra_def_ptr = (EXTRA_DEVICE_SPEC *)definition->extra_data;
1016         if (extra_def_ptr->dab_ok == TRUE) {
1017             dab_info_index = instance->dab_info_index;
1018
1019             dab_ptr = dab_list[dab_info_index];
1020             new_dab_ptr = new_dab_list[dab_info_index];
1021
1022             if ((new_dab_ptr != NULL) &&
1023                 (strcmp(dab_ptr->part_name,
1024                     new_dab_ptr->part_name) == 0) &&
1025                 (strcmp(dab_ptr->location, new_dab_ptr->location) == TRUE)) {
1026                 /* The same DAB is present in the same location */
1027
1028                 if (all_dab_unit_active_and_initialized(dab_ptr) == FALSE) {
1029                     /* Check to see if the DAB is touched (removed and inserted
1030                      * back again) at all by looking at the ACTIVE bit.
1031                     */
1032                     if (definition->device_type == PRIVATE) {
1033                         /* If any of them is touched then invalidate PRIVATE
1034                         * device.
1035                         */
1036                         extra_def_ptr->dab_ok = FALSE;
1037                         instance->dab_info_index = -1;
1038                     }
1039                 }
1040             }
1041             else {
1042                 /* The DAB is removed or changed */
1043                 extra_def_ptr->dab_ok = FALSE;
1044                 instance->dab_info_index = -1;
1045             }
1046         }
1047
1048         /* Increment the DAB_INFO.act_inst_count or DAB_INFO.act_var_count,
1049          * set the DAB_INFO.used_as_private appropriately.
1050         */
1051         if (extra_def_ptr->dab_ok == TRUE) {
1052             if (instance->is_fault == TRUE)
1053                 ++new_dab_ptr->act_var_count;
1054             else {
1055                 ++new_dab_ptr->act_inst_count;
1056                 if (definition->device_type == PRIVATE)
1057                     new_dab_ptr->used_as_private = TRUE;
1058             }
1059         }
1060     }
1061 }
1062
1063 /* We have to reset MAGIC ERRORS for ALL magic chips in the system; the
1064 * good ones and the bad ones.
1065 */
1066 reset_magic_error(new_dab_list);
1067 reset_magic_error(bad_dab_list);
1068
1069 raise_pal_reset();
1070
1071 rls_bad_dab_list();
1072
1073 measure_dut_vcc(new_dab_list);
1074
1075 /* release the old dab list */
1076 for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) {
1077     if (dab_list[slotno] != NULL) {
1078         rls_dab_info(dab_list[slotno]);
1079         dab_list[slotno] = NULL;
1080     }
1081 }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
10/68

```

1081 }
1082 }
1083
1084 DPRINTF(("DAB found: %s"),
1085 /* copy the new dab into dab_list */
1086 for (slotno = 0; slotno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotno) {
1087     if (new_dab_list[slotno] != NULL) {
1088         dab_list[slotno] = new_dab_list[slotno];
1089
1090         DPRINTF(("slotno", dab_list[slotno]->part_name));
1091
1092         /* Turn on IS USE led */
1093         if ((dab_list[slotno]->act_inst_count +
1094             dab_list[slotno]->act_var_count) != 0)
1095             turn_on_is_use(dab_list[slotno]);
1096     }
1097 }
1098
1099 clear_non_fatal_mod_err();
1100
1101 if (asy_pel_with_reset_asserted() == TRUE) {
1102     if (again_count++ < 10) {
1103         DPRINTF(("some pels still have reset asserted\n"));
1104         goto again;
1105     }
1106     else {
1107         DPRINTF(("infinite loop detected in reconfigure_dab. Bailing out...\n"));
1108     }
1109 }
1110
1111 DPRINTF(("exiting reconfigure_dab\n"));
1112 }
1113
1114 asy_pel_with_reset_asserted()
1115 {
1116     u_long   addr;
1117     u_short  word;
1118     u_char   lane_no;
1119     u_char   slotno;
1120
1121     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
1122         for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) {
1123             if (system_config->lane[lane_no]->pel[slotno] == NULL)
1124                 continue;
1125
1126             addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +
1127                 LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
1128                 PEL_STATUS_CONTROL_OFFSET;
1129
1130             word = read_lec_long((u_long *)addr);
1131             DPRINTF(("apera: ls: %d sl: %d pel status: %04x\n",
1132                 lane_no, slotno, word));
1133
1134             if ((word & "PEL_CS_RESET_MASK" == 0) {
1135                 DPRINTF(("pel reset on lane: %d slot: %d still asserted\n",
1136                     lane_no, slotno));
1137                 return(TRUE);
1138             }
1139         }
1140     }
1141     return(FALSE);
1142 }
1143
1144 cmp_dab_location(dab_ptr1, dab_ptr2)
1145 DAB_INFO *dab_ptr1;
1146 DAB_INFO *dab_ptr2;
1147 {
1148     /* compare the each unit position of dab_ptr1 and dab_ptr2.
1149     * Return TRUE if they are the same else return FALSE.
1150     */
1151
1152     u_char   unitno;
1153     u_char   total_unit;
1154
1155     if (dab_ptr1->unit_count != dab_ptr2->unit_count)
1156         return(FALSE);
1157
1158     if (dab_ptr1->unit_count_per_lane != dab_ptr2->unit_count_per_lane)
1159         return(FALSE);
1160
1161     total_unit = dab_ptr1->unit_count;
1162     for (unitno = 0; unitno < total_unit; ++unitno) {
1163         if ((dab_ptr1->unit_location[unitno].lane_no !=
1164             dab_ptr2->unit_location[unitno].lane_no) ||
1165             (dab_ptr1->unit_location[unitno].slot_no !=
1166             dab_ptr2->unit_location[unitno].slot_no))
1167             return(FALSE);
1168     }
1169     return(TRUE);
1170 }
1171
1172 cmp_dab_shape(dab_ptr1, dab_ptr2)
1173 DAB_INFO *dab_ptr1;
1174 DAB_INFO *dab_ptr2;
1175 {
1176     /* compare the each unit relative position of dab_ptr1 and dab_ptr2.
1177     * Return TRUE if they are the same else return FALSE.
1178     */
1179
1180     u_char   unitno;
1181     u_char   total_unit;
1182     u_char   lanesol;
1183     u_char   slotsol;
1184     u_char   lanesol2;
1185     u_char   slotsol2;
1186
1187     if (dab_ptr1->unit_count != dab_ptr2->unit_count)
1188         return(FALSE);
1189
1190     if (dab_ptr1->unit_count_per_lane != dab_ptr2->unit_count_per_lane)
1191         return(FALSE);
1192
1193     lanesol = dab_ptr1->unit_location[0].lane_no;
1194     lanesol2 = dab_ptr2->unit_location[0].lane_no;
1195     slotsol = dab_ptr1->unit_location[0].slot_no;
1196     slotsol2 = dab_ptr2->unit_location[0].slot_no;
1197
1198     total_unit = dab_ptr1->unit_count;
1199     for (unitno = 0; unitno < total_unit; ++unitno) {
1200

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/hwconfig.c	DATE 5/23/89	PAGE # 11/69
LINE #	SOURCE TEXT			
1201	if ((dab_ptr->unit_location(unitno).lane_no - lanesol !=			
1202	dab_ptr->unit_location(unitno).lane_no - lanesol2)			
1203	(dab_ptr->unit_location(unitno).slot_no - slotsol !=			
1204	dab_ptr->unit_location(unitno).slot_no - slotsol2))			
1205	return("SE");			
1206	}			
1207	}			
1208	return(TRUE);			
1209	}			
1210	measure_dut_vcc(loc_dab_list)			
1211	DAB_INFO *loc_dab_list;			
1212	{			
1213	DAB_INFO *dab_ptr;			
1214	u_long addr;			
1215	u_long delay;			
1216	u_char lanesol;			
1217	u_char slotsol;			
1218	u_char lanesol2;			
1219	u_char slotsol2;			
1220	u_char value;			
1221	{			
1222	for (dabno = 0; dabno < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++dabno) {			
1223	{			
1224	if (loc_dab_list[dabno] == NULL)			
1225	continue;			
1226	dab_ptr = loc_dab_list[dabno];			
1227	{			
1228	lanesol = dab_ptr->unit_location(0).lane_no;			
1229	slotsol = dab_ptr->unit_location(0).slot_no;			
1230	{			
1231	addr = LANE_0_START_ADDR + lanesol * LANE_ADDR_INC +			
1232	LANE_PEL_0_START_OFFSET + slotsol * LANE_PEL_ADDR_INC +			
1233	PEL_DOTVCC_DAC_READ_OFFSET;			
1234	{			
1235	/* The first measurement is always wrong after power up */			
1236	#ifdef DEBUG			
1237	value = read_loc_long((u_long *)addr) & 0xff;			
1238	delay = 1000;			
1239	while (--delay) {			
1240	value = read_loc_long((u_long *)addr) & 0xff;			
1241	}			
1242	#else			
1243	value = 0x0;			
1244	#endif			
1245	{			
1246	dab_ptr->dut_vcc_measured = MAX_DOT_VCC_VOLTAGE * value / 256;			
1247	{			
1248	DPRINTF(("dut measured: %d\n", dab_ptr->dut_vcc_measured));			
1249	}			
1250	}			
1251	}			
1252	init_bad_dab_list()			
1253	{			
1254	u_char slotsol;			
1255	{			
1256	for (slotsol = 0; slotsol < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotsol)			
1257	bad_dab_list[slotsol] = NULL;			
1258	{			
1259	}			
1260	rle_bad_dab_list()			
1261	{			
1262	u_char slotsol;			
1263	for (slotsol = 0; slotsol < (MAX_LANE_COUNT * MAX_SLOT_COUNT); ++slotsol) {			
1264	if (bad_dab_list[slotsol] == NULL)			
1265	continue;			
1266	{			
1267	rle_dab_info(bad_dab_list[slotsol]);			
1268	{			
1269	bad_dab_list[slotsol] = NULL;			
1270	}			
1271	}			
1272	{			
1273	all_dab_unit_active_and_initialized(dab_ptr)			
1274	DAB_INFO *dab_ptr;			
1275	{			
1276	/* Returns TRUE if all unit of DAB has the ACTIVE bit, the			
1277	INITIALIZE bit set, and RESET is deasserted. The INITIALIZE bit			
1278	is set to 0 when we are labeling the DAB.			
1279	{			
1280	u_long addr;			
1281	u_short value;			
1282	u_char total_unit;			
1283	u_char unitno;			
1284	u_char lanesol;			
1285	u_char slotsol;			
1286	{			
1287	total_unit = dab_ptr->unit_count;			
1288	for (unitno = 0; unitno < total_unit; ++unitno) {			
1289	lanesol = dab_ptr->unit_location(unitno).lane_no;			
1290	slotsol = dab_ptr->unit_location(unitno).slot_no;			
1291	{			
1292	addr = LANE_0_START_ADDR + lanesol * LANE_ADDR_INC +			
1293	LANE_PEL_0_START_OFFSET + slotsol * LANE_PEL_ADDR_INC +			
1294	PEL_STATUS_CONTROL_OFFSET;			
1295	{			
1296	value = (u_short)read_loc_long((u_long *)addr);			
1297	DPRINTF(("adusai: ls: %d sl: %d pel status: %04x\n",			
1298	lanesol, slotsol, value));			
1299	{			
1300	if (((value & PEL_CS_ACTIVE_MASK) == 0)			
1301	((value & PEL_CS_INITIALIZE_MASK) == 0)			
1302	((value & PEL_CS_RESET_MASK) == 0))			
1303	return(FALSE);			
1304	{			
1305	return(TRUE);			
1306	{			
1307	}			
1308	{			
1309	configure_pam(lanesol, lane_ptr)			
1310	u_char lanesol;			
1311	LANE_INFO *lane_ptr;			
1312	{			
1313	PAM_INFO *pam_ptr;			
1314	u_long board_id;			
1315	#ifdef MODELER			
1316	u_long addr;			
1317	u_long link_addr;			
1318	long i;			
1319	#endif			
1320	u_long total_mem;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
12/70

```

1321 u_char      pamno;
1322 u_char      temp;
1323 u_char      pam_count;
1324 u_char      config;
1325 u_char      pam_error_encountered;
1326 u_char      pam_type;
1327 u_char      cur_pam_type;
1328
1329 DPRINTF(("inside configure_pam\n"));
1330 cur_pam_type = PAC_PAM_2M;
1331
1332 (void)read_pam_count_reg(laseno, &pam_count);
1333 if (pam_count == 0) {
1334     lm_config_error(CERR_NO_PAM_FOR_PAC, laseno, 0);
1335     DPRINTF(("ERROR: no pam on lase %d\n", laseno));
1336     return(FAILURE);
1337 }
1338
1339 DPRINTF(("pam count: %d\n", pam_count));
1340
1341 /* find all of the PAM's in this lase */
1342 pam_error_encountered = 0;
1343 total_mem = 0;
1344 for (pamno = 0; pamno < pam_count; ++pamno) {
1345     (void)read_pam_present(laseno, pamno, &temp);
1346     if (temp) {
1347         DPRINTF(("found pam: %d\n", pamno));
1348
1349         /* The PAM is present */
1350         pam_ptr = new_pam_info();
1351         pam_ptr->pam[pamno] = pam_ptr;
1352         (void)read_pam_id(laseno, pamno, &pam_type, &board_id);
1353
1354         pam_ptr->board_id = board_id;
1355
1356         /* Check to make sure that the PAM's are ordered according
1357          * to decreasing memory size.
1358          */
1359         if (pam_type > cur_pam_type) {
1360             lm_config_error(CERR_PAM_STACKED_WRONG, laseno, pamno);
1361             DPRINTF(("ERROR: PAM %d is bigger than previous PAM\n",
1362                 pamno));
1363             pam_error_encountered = 1;
1364         }
1365         else
1366             cur_pam_type = pam_type;
1367
1368         switch (pam_type) {
1369             case PAC_PAM_128K:
1370                 pam_ptr->mem_size = PAM128K_PTRN_COUNT;
1371                 break;
1372             case PAC_PAM_512K:
1373                 pam_ptr->mem_size = PAM512K_PTRN_COUNT;
1374                 break;
1375             case PAC_PAM_2M:
1376                 pam_ptr->mem_size = PAM2M_PTRN_COUNT;
1377                 break;
1378             default:
1379                 break;
1380         }
1381
1382         DPRINTF(("pam size: %d\n", pam_ptr->mem_size));
1383         total_mem += pam_ptr->mem_size;
1384     }
1385     else {
1386         lm_config_error(CERR_PAM_STRAPPED_WRONG, laseno, pamno);
1387         DPRINTF(("ERROR: PAM %d on lase %d is strapped incorrectly\n",
1388             pamno, laseno));
1389         pam_error_encountered = 1;
1390     }
1391 }
1392
1393 if (pam_error_encountered) {
1394     /* If there are any problems with PAM strapping, remove
1395     * all of the PAMs from the list.
1396     */
1397     for (pamno = 0; pamno < MAX_PAM_COUNT; ++pamno)
1398         pam_ptr->pam[pamno] = NULL;
1399     return(FAILURE);
1400 }
1401
1402 /* Write PAM config register */
1403 switch (pam_count) {
1404     case 1:
1405         switch (total_mem / PTRN_PER_BLOCK) {
1406             case BLOCKS_IN_128K:
1407                 config = 60;
1408                 break;
1409             case BLOCKS_IN_512K:
1410                 config = 61;
1411                 break;
1412             case BLOCKS_IN_2M:
1413                 config = 62;
1414                 break;
1415             default:
1416                 break;
1417         }
1418     case 2:
1419         config = ((total_mem / PTRN_PER_BLOCK) / BLOCKS_IN_128K) - 1;
1420         break;
1421     case 3:
1422         break;
1423     case 4:
1424         break;
1425     default:
1426         /* This condition will never occur */
1427         break;
1428 }
1429
1430 /* set ODD parity on LOW and HIGH word --> set the bit to 1 */
1431 config |= PAC_CONFIG_HIGH_WORD_PARITY_MASK |
1432          PAC_CONFIG_LOW_WORD_PARITY_MASK;
1433
1434 DPRINTF(("PAC config reg on lase: %d --> %08x\n", laseno, config));
1435
1436 /* Configure the PAC with EVEN parity, then clear the memory, and
1437 * then set it to ODD parity. This will catch software problems; i.e.
1438 * if the software reads uninitialized memory inadvertently.
1439 */

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/hwconfig.c	DATE 5/23/89 TIME 6:14:42 pm	PAGE # 13/71
LINE #	SOURCE TEXT			
1441	write_loc_long((u_long *)(&lane_offset(laneno) +			
1442	LANE_PAC_START_OFFSET + PAC_PAM_CONFIG_REG_OFFSET),			
1443	(u_long)config);			
1444				
1445	#ifdef MODELER			
1446	addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC;			
1447	link_addr = addr + LANE_LINK_TABLE_OFFSET;			
1448				
1449	for (i = 0; i < total_num; ++i) {			
1450	write_loc_long((u_long *)(&addr), (u_long)0);			
1451	write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET), (u_long)0);			
1452	write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET), (u_long)0);			
1453	addr += PTHN_ADDR_INC;			
1454	}			
1455				
1456	for (i = 0; i < MAX_LINK_TABLE_COUNT; ++i) {			
1457	write_loc_long((u_long *)(&link_addr), (u_long)0);			
1458	link_addr += LINK_TABLE_ADDR_INC;			
1459	}			
1460	#else			
1461	#ifdef DIRECTCONN			
1462	dc_clear_ptr(laneno, (u_long)total_num);			
1463	#endif			
1464	#endif			
1465				
1466	/* set ODD parity on LOW and HIGH word --> set the bit to 1 */			
1467	config  = "PAC_CONFIG_HIGH_WORD_PARITY_MASK			
1468	PAC_CONFIG_LOW_WORD_PARITY_MASK;			
1469				
1470	write_loc_long((u_long *)(&lane_offset(laneno) +			
1471	LANE_PAC_START_OFFSET + PAC_PAM_CONFIG_REG_OFFSET),			
1472	(u_long)config);			
1473				
1474	DPRINTF(("exiting configure_pam\n"));			
1475	return(SUCCESS);			
1476	}			
1477				
1478	raise_pel_reset()			
1479	{			
1480	u_long magic_addr;			
1481	u_long addr;			
1482	u_short word;			
1483	u_char laneno;			
1484	u_char slotno;			
1485				
1486	/* This routine doesnot RESET on all PELS which still have			
1487	* reset asserted */			
1488	for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) {			
1489				
1490	for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno) {			
1491	if (system_config->lane(laneno)->pel(slotno) == NULL)			
1492	continue;			
1493				
1494	magic_addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +			
1495	LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +			
1496	PEL_NC_0_OFFSET;			
1497				
1498	(void)read_loc_long((u_long *)(&magic_addr + NC_RESET_REG_OFFSET));			
1499				
1500	/* Need to read some other Magic Register to affect reset */			
1501	(void)read_loc_long((u_long *)(&magic_addr + NC_DATA_OUT_REG_OFFSET));			
1502				
1503	addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +			
1504	LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +			
1505	PEL_STATUS_CONTROL_OFFSET;			
1506				
1507	word = read_loc_long((u_long *)(&addr));			
1508	DPRINTF(("rpr: la: %d sl: %d pel status: %04x\n",			
1509	laneno, slotno, word));			
1510				
1511	if (word & "PEL_CS_RESET_MASK)			
1512	continue;			
1513				
1514	/* assert RESET */			
1515	word  = PEL_CS_RESET_MASK & PEL_CS_IN_USE_LED_MASK;			
1516	write_loc_long((u_long *)(&addr), (u_long)word);			
1517				
1518	/* write INITIALIZE to 0 with RESET -- 1 */			
1519	word = (word & PEL_CS_INITIALIZE_MASK)   "PEL_CS_RESET_MASK;			
1520	write_loc_long((u_long *)(&addr), (u_long)word);			
1521				
1522				
1523	}			
1524				
1525				
1526	check_pel_stacking(dab_ptr, pel_location)			
1527	DAB_INFO -> dab_ptr;			
1528	u_char pel_location[MAX_SLOT_COUNT];			
1529	{			
1530	char i;			
1531	u_char slots_occupied[MAX_LANE_COUNT][MAX_SLOT_COUNT];			
1532	char minlaneno;			
1533	char maxlaneno;			
1534	char minslotno;			
1535	char maxslotno;			
1536	char laneno;			
1537	char slotno;			
1538				
1539	minlaneno = MAX_LANE_COUNT;			
1540	maxlaneno = -1;			
1541	minslotno = MAX_SLOT_COUNT;			
1542	maxslotno = -1;			
1543				
1544	/* Initialize slots_occupied */			
1545	for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno)			
1546	for (slotno = 0; slotno < MAX_SLOT_COUNT; ++slotno)			
1547	slots_occupied[laneno][slotno] = 0;			
1548				
1549	/* Mark all of the slots occupied by this device */			
1550	for (i = 0; i < dab_ptr->unit_count; ++i) {			
1551	laneno = dab_ptr->unit_location[i].lane_no;			
1552	slotno = dab_ptr->unit_location[i].slot_no;			
1553				
1554	slots_occupied[laneno][slotno] = 1;			
1555				
1556	if (laneno < minlaneno)			
1557	minlaneno = laneno;			
1558	if (laneno > maxlaneno)			
1559	maxlaneno = laneno;			
1560	if (slotno < minslotno)			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/hwconfig.c

DATE 5/23/89  
TIME 6:14:42 pm

PAGE #  
14/72

```

LINE # SOURCE TEXT
1561 minslotno = slotno;
1562 if (slotno > maxslotno)
1563     maxslotno = slotno;
1564 }
1565
1566 for (slotno = minslotno; slotno <= maxslotno; ++slotno) {
1567     /* Check if this slot is occupied at all */
1568     for (laneno = minlaneno; laneno <= maxlaneno; ++laneno) {
1569         if (slots_occupied[laneno][slotno] == 1)
1570             break;
1571     }
1572     /* Go to the next slot if there is no unit in this slot */
1573     if (laneno > maxlaneno)
1574         continue;
1575     /* Make sure that at this slotno there is a unit on each lane
1576      * that the device occupies. Don't do this check on
1577      * the rightmost slot.
1578      */
1579     if (slotno != maxslotno) {
1580         for (laneno = minlaneno; laneno <= maxlaneno; ++laneno) {
1581             if (dab_ptr->lane_used[laneno] == FALSE)
1582                 continue;
1583             if (slots_occupied[laneno][slotno] == 0)
1584                 return(FAILURE);
1585         }
1586     }
1587     /* Go through all other lanes occupied by this device and compare
1588      * the PEL stacking on this lane with the PEL stacking on the lane
1589      * found above.
1590      */
1591     for (laneno = minlaneno + 1; laneno <= maxlaneno; ++laneno) {
1592         if (dab_ptr->lane_used[laneno] == FALSE)
1593             continue;
1594         /* This condition can only happen for the rightmost slot */
1595         if (slots_occupied[laneno][slotno] == 0)
1596             continue;
1597         if (check_pel_to_left(pel_location, minlaneno,
1598                               laneno, slotno) == FAILURE) {
1599             return(FAILURE);
1600         }
1601     }
1602     return(SUCCESS);
1603 }
1604
1605 check_pel_to_left(pel_location, laneno1, laneno2, toslotno)
1606 u_char pel_location[[]MAX_SLOT_COUNT];
1607 char laneno1;
1608 char laneno2;
1609 char toslotno;
1610 {
1611     u_char slotno;
1612     DPRINTF(("inside check_pel_to_left on slot: %d between lane %d and %d\n",
1613             toslotno, laneno1, laneno2));
1614     for (slotno = 0; slotno < toslotno; ++slotno) {
1615         if (pel_location[laneno1][slotno] != pel_location[laneno2][slotno])
1616             return(FAILURE);
1617     }
1618     return(SUCCESS);
1619 }
1620
1621 init_magic_and_pel(laneno, slotno)
1622 u_char laneno;
1623 u_char slotno;
1624 {
1625     u_long magic_addr;
1626     DPRINTF(("inside init_magic_and_pel, lane: %d slot: %d\n",
1627             laneno, slotno));
1628     magic_addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +
1629                 LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
1630                 PEL_MC_0_OFFSET;
1631     (void)read_loc_long((u_long *)(&magic_addr + MC_RESET_REG_OFFSET));
1632     /* Need to read some other Magic reg to affect reset */
1633     (void)read_loc_long((u_long *)(&magic_addr + MC_DATA_OUT_REG_OFFSET));
1634     (void)read_loc_long((u_long *)(&magic_addr));
1635     init_dab(laneno, slotno);
1636 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89 PAGE #  
TIME 6:14:43 pm 1/73

```

LINE # SOURCE TEXT
1  /* SCOS_ID: initseq.c rev 3.1. 4/24/89 at 07:53:13 */
2  #include "device.h"
3  #include "message.h"
4  #include "hardware.h"
5  #include "osprun.h"
6  #include "lmserver.h"
7
8  #ifdef DEBUG
9  extern u_char loop_till_key;
10 extern u_long debug_key;
11 extern u_long debug_chan;
12 #endif
13
14 build_static_pattern_seq(instance);
15 INSTANCE_INFO *instance;
16
17 /* This routine builds the static pattern seq for an instance:
18  * PREAMBLE, PRE_FB_SEQ, FB_SEQ, POST_FB_SEQ
19  * by calling load_preamble(), load_pre_feedback_seq(),
20  * load_feedback_seq(), and load_post_feedback_seq(), respectively.
21  * Upon return from the routines load_preamble() and
22  * load_pre_feedback_seq() the INSTANCE.unit_addr is pointing to the
23  * last pattern loaded, whereas the INSTANCE.unit_addr is pointing at
24  * the next pattern address to be loaded upon return from
25  * load_feedback_seq() and load_post_feedback_seq().
26  * The following combinations of preamble.seq and reset seq are allowed:
27  *
28  * PREAMBLE PRE FB POST
29  * PREAMBLE PRE FB
30  * PREAMBLE FB POST
31  * PREAMBLE FB
32  * PREAMBLE PRE
33  *
34  * any other combinations are illegal and should be checked by the parser.
35  */
36
37 PREAMBLE preamble;
38 DAB_INFO *dab_ptr;
39 u_char lanes;
40 long dummy_count;
41 u_long *unit_addr;
42 u_long overflowed_patterns_count;
43 short i;
44 u_char reload_preamble_flag;
45
46 dab_ptr = dab_list(instance->dab_info_index);
47
48 setup_gbl_dummy_ptrs(dab_ptr);
49
50 if (allocate_initial_block(instance) == FAILURE)
51     return;
52
53 /* set the sequence start addresses on each lane */
54 for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes)
55     instance->seq_start_addr[lanes] =
56         instance->lane_addr[lanes].max_addr - BLOCK_ADDR_INC;
57
58 if (instance->definition->fb_seq_len != 0)
59     if (load_dummy_pattern(instance) == FAILURE)
60         return;
61
62 if (load_preamble(instance, (u_char)FALSE,
63     &preamble, &reload_preamble_flag) == FAILURE)
64     return;
65
66 if (instance->definition->pre_seq_len != 0) {
67     if (grow_pattern(instance) == FAILURE)
68         return;
69
70     if (load_pre_feedback_seq(instance) == FAILURE)
71         return;
72 }
73
74 if (instance->definition->fb_seq_len != 0) {
75     /* Add more dummy patterns to make sure that the feedback signal
76     * is quiet before we branch to the feedback sequence.
77     */
78     if (load_dummy_pattern2(instance) == FAILURE)
79         return;
80
81     if (allocate_feedback_block(instance) == FAILURE)
82         return;
83
84     if (load_feedback_seq(instance, &overflowed_patterns_count) == FAILURE)
85         return;
86
87     /* The sequence end bit has to be specified. SEQ_END_LATENCY patterns
88     * before the actual pattern we want to stop at. If the number of
89     * post feedback sequence is less than the SEQ_END_LATENCY, we need
90     * to add some dummy patterns in the beginning of the block after the
91     * feedback block.
92     */
93
94     /* Total number of lane patterns needed for various unit_count_per_lane
95     * (assuming SEQ_END_LATENCY = 6):
96     *
97     * unit_count_per_lane = 1 --> tot_lane_patterns_needed = 6
98     * unit_count_per_lane = 2 --> tot_lane_patterns_needed = 3
99     * unit_count_per_lane = 3 --> tot_lane_patterns_needed = 2
100    * unit_count_per_lane = 4 --> tot_lane_patterns_needed = 2
101    * unit_count_per_lane = 5 --> tot_lane_patterns_needed = 2
102    * unit_count_per_lane = 6 --> tot_lane_patterns_needed = 1
103    * unit_count_per_lane = 7 --> tot_lane_patterns_needed = 1
104    *
105    */
106    if (overflowed_patterns_count < SEQ_END_LATENCY) {
107        dummy_count = SEQ_END_LATENCY / dab_ptr->unit_count_per_lane;
108        if ((dummy_count * dab_ptr->unit_count_per_lane) < SEQ_END_LATENCY)
109            ++dummy_count;
110        dummy_count -= instance->definition->post_seq_len;
111    }
112    else
113        dummy_count = 0;
114
115    for (i = 0; i < dummy_count; ++i) {
116        write_pattern(instance,
117            instance->unit_addr(instance->cur_unit_addr_index)[0],
118            gbl_dummy_ptr);
119    }
120

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE	5/23/89	PAGE #
TIME	6:14:43 pm	2/74

```

LINE #          SOURCE TEXT
121         if (grow_pattern(instance) == FAILURE)
122             return;
123     }
124     else {
125         /* load_pre_feedback_seq() leaves the unit_addr pointing at the
126          * last pattern written. We need to increment it by one pattern
127          * so that the entry condition is load_post_feedback_seq() is
128          * the same whether we have feedback sequence or not.
129          */
130         if (grow_pattern(instance) == FAILURE)
131             return;
132     }
133     if (instance->definition->post_seq_len != 0)
134         if (load_post_feedback_seq(instance) == FAILURE)
135             return;
136     if (reload_preamble_flag == TRUE) {
137         if (reload_preamble(instance, &preamble) == FAILURE) {
138             free_preamble(&preamble);
139             return;
140         }
141     }
142     free_preamble(&preamble);
143     if (set_initial_values(instance) == FAILURE)
144         return;
145     /* Copy the current pattern_count to the static pattern_count.
146      * Note that grow_pattern() was called last, so the number is the
147      * pattern_count is 1 logical pattern more than what it should be.
148      */
149     instance->static_pattern_count = instance->pattern_count -
150                                     dab_ptr->unit_count_per_lane;
151     /* Copy the unit_addr to first_wor_ptr unit_addr */
152     temp_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0];
153     for (i = 0; i < dab_ptr->unit_count; ++i)
154         instance->first_wor_ptr_unit_addr[i] = temp_unit_addr[i];
155 }
156
157 load_preamble(instance, preamble_for_power_up, preamble, reload_preamble_flag)
158 INSTANCE_INFO *instance;
159 u_char          preamble_for_power_up;
160 PREAMBLE        *preamble;
161 u_char          *reload_preamble_flag;
162 {
163     PREAMBLE_WORD *preamble_word;
164     PTRN_BITS_LONGWORD *unit_ptrn;
165     PTRN_BITS_LONGWORD *preamble_unit_ptrn;
166     PTRN_BITS *ptrn_bits_ptr;
167     PIN_SPEC *pin_spec_ptr;
168     DAB_INFO *dab_ptr;
169     DEVICE_SPEC *dev_ptr;
170     EXTRA_DEVICE_SPEC *extra_dev_ptr;
171     SEQ_SPEC *seq_spec_ptr;
172     PTRN_BITS_LONGWORD *ident_outputs_ptr;
173     PTRN_BITS_LONGWORD *ident_ios_ptr;
174     u_long *seq_ptr;
175     u_long *temp_unit_addr;
176     u_short seq_count;
177     u_short pin_count;
178     u_short pinno;
179     u_short unitno;
180     u_char wordno;
181     u_char hitno;
182     u_char unit_count;
183     u_char i;
184     u_char j;
185     u_char temp;
186
187     DPRINTF(("inside load_preamble\n"));
188     *reload_preamble_flag = FALSE;
189
190     def_ptr = instance->definition;
191     extra_dev_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
192
193     dab_ptr = dab_list[instance->dab_info_index];
194     unit_count = dab_ptr->unit_count;
195
196     preamble_word = (PREAMBLE_WORD *)preamble;
197
198     for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *); ++i) {
199         preamble_word->word[i] =
200             (PTRN_BITS *)DCALLOC((unsigned)unit_count,
201                                   (unsigned)sizeof(PTRN_BITS));
202
203         if (preamble_word->word[i] == NULL) {
204             is_queue_message(ERROR_MSG, "out of memory on modeler");
205             for (j = 0; j < i; ++j) {
206                 DFREE((char *)preamble_word->word[j]);
207             }
208             return(FAILURE);
209         }
210     }
211
212     for (i = 0; i < unit_count; ++i) {
213         unit_ptrn = (PTRN_BITS_LONGWORD *)preamble->adlen3b[i];
214         unit_ptrn->word[0] = 0xffffffff;
215         unit_ptrn->word[1] = 0xffffffff;
216         unit_ptrn->word[2] = 0xffff0000;
217         unit_ptrn = (PTRN_BITS_LONGWORD *)preamble->adlen2b[i];
218         unit_ptrn->word[0] = 0xffffffff;
219         unit_ptrn->word[1] = 0xffffffff;
220         unit_ptrn->word[2] = 0xffff0000;
221         unit_ptrn = (PTRN_BITS_LONGWORD *)preamble->adlen1b[i];
222         unit_ptrn->word[0] = 0xffffffff;
223         unit_ptrn->word[1] = 0xffffffff;
224         unit_ptrn->word[2] = 0xffff0000;
225         unit_ptrn = (PTRN_BITS_LONGWORD *)preamble->seqqdb[i];
226         unit_ptrn->word[0] = 0xffffffff;
227         unit_ptrn->word[1] = 0xffffffff;
228         unit_ptrn->word[2] = 0xffff0000;
229         unit_ptrn = (PTRN_BITS_LONGWORD *)preamble->lcyqdb[i];
230         unit_ptrn->word[0] = 0xffffffff;
231     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE	5/23/89	PAGE #
TIME	6:14:43 pm	3/75

```

LINE #
241      unit_ptr->word[1] = 0xffffffff;
242      unit_ptr->word[2] = 0xffffffff;
243      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->lcymdb[1];
244      unit_ptr->word[0] = 0xffffffff;
245      unit_ptr->word[1] = 0xffffffff;
246      unit_ptr->word[2] = 0xffffffff;
247      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->format1[1];
248      unit_ptr->word[0] = 0xffffffff;
249      unit_ptr->word[1] = 0xffffffff;
250      unit_ptr->word[2] = 0xffffffff;
251      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->format0[1];
252      unit_ptr->word[0] = 0xffffffff;
253      unit_ptr->word[1] = 0xffffffff;
254      unit_ptr->word[2] = 0xffffffff;
255      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->dummy_hmdeab[1];
256      unit_ptr->word[0] = 0xffffffff;
257      unit_ptr->word[1] = 0xffffffff;
258      unit_ptr->word[2] = 0xffffffff;
259      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->hmdenb[1];
260      unit_ptr->word[0] = 0xffffffff;
261      unit_ptr->word[1] = 0xffffffff;
262      unit_ptr->word[2] = 0xffffffff;
263      unit_ptr = (PTRN_BITS_LONGWORD *) &preamble->dummy_data2[1];
264      unit_ptr->word[0] = 0xffffffff;
265      unit_ptr->word[1] = 0xffffffff;
266      unit_ptr->word[2] = 0xffffffff;
267  }
268
269  pin_count = def_ptr->pin_cnt;
270  pin_spec_ptr = &def_ptr->pin_table[0];
271  for (pinno = 0; pinno < pin_count; ++pinno) {
272
273      if (pin_spec_ptr->direction == NONE) {
274          ++pin_spec_ptr;
275          continue;
276      }
277
278      if ((pin_spec_ptr->direction == POWER) ||
279          (pin_spec_ptr->direction == GROUND) ||
280          (pin_spec_ptr->direction == NC)) {
281
282          /* Float these pins */
283
284          word_ptr = &pin_spec_ptr->word;
285          unitno = word_ptr->unitno;
286          wordno = word_ptr->wordno;
287          bitno = word_ptr->bitno;
288
289          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
290          set_ptrn_bit(&preamble->adlshdb [unitno], wordno, bitno);
291
292          ++pin_spec_ptr;
293          continue;
294      }
295
296      /* INPUT, OUTPUT, or IO pin */
297
298      word_ptr = &pin_spec_ptr->word;
299      unitno = word_ptr->unitno;
300      wordno = word_ptr->wordno;
301      bitno = word_ptr->bitno;
302
303      /* HUBBIS */
304      if (pin_spec_ptr->h_drive == ALWAYS_ON)
305          set_ptrn_bit(&preamble->hddiab[unitno], wordno, bitno);
306
307      /* For I/O STORE pins in NON ULTRA FAST mode, set HUBBIS to 1.
308       * We can tell if it's in Non-Ultra-Fast mode because INREQD
309       * will be off.
310       */
311      if ((pin_spec_ptr->direction == IO) &&
312          (pin_spec_ptr->pin_class == STORE) &&
313          (pin_spec_ptr->pin_seq_drive != H_DRIVE) &&
314          (pin_spec_ptr->pin_seq_drive != EM_DRIVE)) {
315          set_ptrn_bit(&preamble->hddiab[unitno], wordno, bitno);
316      }
317
318      /* Set DOFF to 1 (edge 5) for I/O STORE pins */
319      if ((pin_spec_ptr->direction == IO) &&
320          (pin_spec_ptr->pin_class == STORE))
321          set_ptrn_bit(&preamble->doff[unitno], wordno, bitno);
322
323      /* HMDENB */
324      if (pin_spec_ptr->h_drive == DRIVE_ON)
325          reset_ptrn_bit(&preamble->hmdenb[unitno], wordno, bitno);
326
327      /* HMDENB and SEQHDB */
328      switch (pin_spec_ptr->pin_seq_drive) {
329      case NO_DRIVE:
330          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
331          break;
332      case H_DRIVE:
333          reset_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
334          set_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
335          break;
336      case M_DRIVE:
337          /* Default is Medium drive */
338          break;
339      case EM_DRIVE:
340          reset_ptrn_bit(&preamble->seqhdb [unitno], wordno, bitno);
341          break;
342      default:
343          break;
344      }
345
346      /* LCYCHDB and LCTCHDB */
347      switch (pin_spec_ptr->last_cyc_drive) {
348      case NO_DRIVE:
349          break;
350      case H_DRIVE:
351          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
352          break;
353      case M_DRIVE:
354          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
355          break;
356      case EM_DRIVE:
357          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
358          reset_ptrn_bit(&preamble->lcychdb [unitno], wordno, bitno);
359          break;
360      default:

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89  
TIME 6:14:43 pm

PAGE #  
4/76

LINE # SOURCE TEXT

```

361     break;
362 }
363
364 /* FORWARD[1-0], SET[1-0], RESET[1-0] */
365 switch (pin_spec_ptr->clk_format) {
366     case DNRZ:
367         if (pin_spec_ptr->pin_class == STORE) {
368             /* use edge #1:
369              * SET [1-0] <- 1
370              * RESET [1-0] <- 3
371              */
372             set_ptrn_bit(&preamble->set0 [unitno], wordno, bitno);
373             set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
374             set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
375         }
376         else {
377             /* use edge #2:
378              * SET [1-0] <- 0
379              * RESET [1-0] <- 2
380              */
381             set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
382         }
383     }
384     break;
385     case R0:
386         reset_ptrn_bit(&preamble->format1 [unitno], wordno, bitno);
387         /* falling edge (significant edge) --> use edge #3
388          * rising edge (non significant edge) --> use edge #4
389          * SET [1-0] <- 2
390          * RESET [1-0] <- 3
391          */
392         set_ptrn_bit(&preamble->set1 [unitno], wordno, bitno);
393         set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
394         set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
395     }
396     break;
397     case R1:
398         /* Make sure the clock does not toggle in dummy_data */
399         set_ptrn_bit(&preamble->dummy_data [unitno], wordno, bitno);
400         set_ptrn_bit(&preamble->dummy_data2 [unitno], wordno, bitno);
401         reset_ptrn_bit(&preamble->format0 [unitno], wordno, bitno);
402         /* rising edge (significant edge) --> use edge #3
403          * falling edge (non significant edge) --> use edge #1
404          * SET [1-0] <- 1
405          * RESET [1-0] <- 1
406          */
407         set_ptrn_bit(&preamble->set0 [unitno], wordno, bitno);
408         set_ptrn_bit(&preamble->reset0 [unitno], wordno, bitno);
409     }
410     break;
411     case RC:
412         /* ??? not supported */
413         set_ptrn_bit(&preamble->reset1 [unitno], wordno, bitno);
414     }
415     break;
416     default:
417         break;
418 }
419
420 /* SDLEN[3-0] */
421 temp = pin_spec_ptr->a_drive_hi;
422 DPRINTF(("a_drive_high: %d\n", temp));
423 if (temp & 0x8) {
424     set_ptrn_bit(&preamble->adlen3 [unitno], wordno, bitno);
425 }
426 else {
427     reset_ptrn_bit(&preamble->adlen3 [unitno], wordno, bitno);
428 }
429 if (temp & 0x4) {
430     set_ptrn_bit(&preamble->adlen2 [unitno], wordno, bitno);
431 }
432 else {
433     reset_ptrn_bit(&preamble->adlen2 [unitno], wordno, bitno);
434 }
435 if (temp & 0x2) {
436     set_ptrn_bit(&preamble->adlen1 [unitno], wordno, bitno);
437 }
438 else {
439     reset_ptrn_bit(&preamble->adlen1 [unitno], wordno, bitno);
440 }
441 if (temp & 0x1) {
442     set_ptrn_bit(&preamble->adlen0 [unitno], wordno, bitno);
443 }
444 else {
445     reset_ptrn_bit(&preamble->adlen0 [unitno], wordno, bitno);
446 }
447
448 /* SDLEN[3-0]B */
449 temp = pin_spec_ptr->a_drive_low;
450 DPRINTF(("a_drive_low: %d\n", temp));
451 if (temp & 0x8) {
452     reset_ptrn_bit(&preamble->adlen3b [unitno], wordno, bitno);
453 }
454 else {
455     set_ptrn_bit(&preamble->adlen3b [unitno], wordno, bitno);
456 }
457 if (temp & 0x4) {
458     reset_ptrn_bit(&preamble->adlen2b [unitno], wordno, bitno);
459 }
460 else {
461     set_ptrn_bit(&preamble->adlen2b [unitno], wordno, bitno);
462 }
463 if (temp & 0x2) {
464     reset_ptrn_bit(&preamble->adlen1b [unitno], wordno, bitno);
465 }
466 else {
467     set_ptrn_bit(&preamble->adlen1b [unitno], wordno, bitno);
468 }
469 if (temp & 0x1) {
470     reset_ptrn_bit(&preamble->adlen0b [unitno], wordno, bitno);
471 }
472 else {
473     set_ptrn_bit(&preamble->adlen0b [unitno], wordno, bitno);
474 }
475
476 /* Simulate pull up on feedback pin */
477 if (preamble_for_power_up == FALSE) {
478     if (pin_spec_ptr->feedback == 1) {
479         if (pin_spec_ptr->direction == IO) {
480             if (pin_spec_ptr->in_seq_drive != M_DRIVE) {
481                 set_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
482                 reset_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
483                 *reload_preamble_flag = TRUE;
484             }
485         }
486         else if (pin_spec_ptr->direction == OUT) {
487             reset_ptrn_bit(&preamble->seqfdb [unitno], wordno, bitno);
488             *reload_preamble_flag = TRUE;
489         }
490     }
491 }

```

596  
597  
598  
599

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE	5/23/89	PAGE #
TIME	6:14:43 pm	6/78

```

LINE # SOURCE TEXT
601
602 if (!preamble_word->word[i] == (preamble->lcychdb) {
603
604     unit_ptrn = (PTRN_BITS_LONGWORD *)instance->lcychdb_loaded;
605     preamble_unit_ptrn =
606         (PTRN_BITS_LONGWORD *)preamble_word->word[i];
607
608     temp_unit_addr = instance->
609         unit_addr(instance->cur_unit_addr_index)[0];
610
611     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
612
613         /* copy the addr of LCYCHDB into lcychdb_addr in instance_info */
614         instance->lcychdb_addr(unitno) = temp_unit_addr(unitno);
615
616         /* copy the LCYCHDB pattern into lcychdb_loaded in instance_info */
617         for (wordno = 0; wordno < 3; ++wordno) {
618             unit_ptrn[unitno].word[wordno] =
619                 preamble_unit_ptrn[unitno].word[wordno];
620         }
621     }
622
623 }
624
625 if (!preamble_word->word[i] == (preamble->lcycmdb) {
626
627     unit_ptrn = (PTRN_BITS_LONGWORD *)instance->lcycmdb_loaded;
628     preamble_unit_ptrn =
629         (PTRN_BITS_LONGWORD *)preamble_word->word[i];
630
631     temp_unit_addr = instance->
632         unit_addr(instance->cur_unit_addr_index)[0];
633
634     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
635
636         /* copy the addr of LCYCHDB into lcycmdb_addr in instance_info */
637         instance->lcycmdb_addr(unitno) = temp_unit_addr(unitno);
638
639         /* copy the LCYCHDB pattern into lcycmdb_loaded in instance_info */
640         for (wordno = 0; wordno < 3; ++wordno) {
641             unit_ptrn[unitno].word[wordno] =
642                 preamble_unit_ptrn[unitno].word[wordno];
643         }
644     }
645
646 }
647
648 write_patterns(instance,
649               instance->unit_addr(instance->cur_unit_addr_index)[0],
650               preamble_word->word[i]);
651
652 /* Don't grow the pattern on the last preamble pattern,
653  * because otherwise grow_patterns() might allocate new blocks which
654  * will have to be retracted if the preamble is followed by feedback
655  * sequence (NO pre feedback sequence).
656  */
657 if (i != ((sizeof(PREAMBLE) / sizeof(PTRN_BITS *)) - 1))
658     if (grow_patterns(instance) == FAILURE)
659         return(FAILURE);
660
661 return(SUCCESS);
662 }
663
664 free_preamble(preamble);
665 PREAMBLE *preamble;
666 {
667     PREAMBLE_WORD *preamble_word,
668     u_char i;
669
670     preamble_word = (PREAMBLE_WORD *)preamble;
671
672     for (i = 0; i < sizeof(PREAMBLE) / sizeof(PTRN_BITS *); ++i)
673         FREE((char *)preamble_word->word[i]);
674 }
675
676 reload_preamble(instance, preamble)
677 INSTANCE_INFO *instance,
678 PREAMBLE *preamble,
679 {
680     PREAMBLE_WORD *preamble_word,
681     PTRN_BITS_LONGWORD *unit_ptrn,
682     PTRN_BITS_LONGWORD *preamble_unit_ptrn,
683     PIN_SPEC *pin_spec_ptr,
684     UWB_OFFSET *uwb_ptr,
685     DEVICE_SPEC *dev_ptr,
686     DAB_INFO *dab_ptr,
687     u_long pin_count,
688     u_short temp_unit_addr,
689     u_short pinno,
690     u_short unitno,
691     u_char wordno,
692     u_char bitno,
693     i;
694
695     DPRINTF(("inside reload_preamble\n"));
696     dev_ptr = instance->definition;
697     dab_ptr = dab_list(instance->dab_info_index);
698
699     preamble_word = (PREAMBLE_WORD *)preamble;
700
701     /* Load a dummy pattern after the feedback/post-feedback sequence
702     * to let the last DATA pattern to get out of the MAGIC chip
703     * before reloading the preamble.
704     */
705     setup_gbl_dummy_ptrn(dab_ptr);
706     write_patterns(instance,
707                   instance->unit_addr(instance->cur_unit_addr_index)[0],
708                   gbl_dummy_ptrn);
709     if (grow_patterns(instance) == FAILURE)
710         return(FAILURE);
711
712     pin_count = dev_ptr->pin_cnt;
713     pin_spec_ptr = dev_ptr->pin_table[0];
714     for (pinno = 0; pinno < pin_count; ++pinno) {
715         if ((pin_spec_ptr->direction == NONE) ||
716             (pin_spec_ptr->direction == POWER) ||
717             (pin_spec_ptr->direction == GROUND))
718             continue;
719     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/initseq.c	DATE 5/23/89 TIME 6:14:43 pm	PAGE # 7/79
LINE #	SOURCE TEXT			
721	<pre>         (pin_spec_ptr-&gt;direction == MC)            (pin_spec_ptr-&gt;direction == IN)) {         *pin_spec_ptr;         continue;     }      wmb_ptr = wmb_ptr + short_offset(pins);     unitno = wmb_ptr-&gt;unitno;     wordno = wmb_ptr-&gt;wordno;     bitno = wmb_ptr-&gt;bitno;      /* SUCCEEDS and SUCCEEDS */     switch (pin_spec_ptr-&gt;is_seq_drive) {     case NO_DRIVE:         set_ptrn_bit(&amp;preamble-&gt;seqhdb [unitno], wordno, bitno);         break;     case S_DRIVE:         reset_ptrn_bit(&amp;preamble-&gt;seqhdb [unitno], wordno, bitno);         set_ptrn_bit(&amp;preamble-&gt;seqhdb [unitno], wordno, bitno);         break;     case M_DRIVE:         /* Default is Medium drive */         break;     case B_DRIVE:         reset_ptrn_bit(&amp;preamble-&gt;seqhdb [unitno], wordno, bitno);         break;     default:         break;     }      *pin_spec_ptr; }  for (i = 0; i &lt; sizeof(PREAMBLE) / sizeof(PTRN_BITS *); ++i) {     /* Don't load the first half of the preamble */     if ((i % 2) &amp; 1) continue;     if ((i % 2) &amp; 1) {         /* Break out of the for loop after we load seqhdb */         if (&amp;preamble-&gt;word[i] == &amp;preamble-&gt;dummy_hndb) {             break;         }         if (&amp;preamble-&gt;word[i] == &amp;preamble-&gt;lcychdb) {             unit_ptrn = (PTRN_BITS_LONGWORD *)instance-&gt;lcychdb_loaded;             preamble_unit_ptrn =                 (PTRN_BITS_LONGWORD *)preamble-&gt;word[i];             temp_unit_addr = instance-&gt;                 unit_addr(instance-&gt;cur_unit_addr_index)[0];             for (unitno = 0; unitno &lt; dab_ptr-&gt;unit_count; ++unitno) {                 /* copy the addr of LCTCHDB into lcychdb_addr in instance_info */                 instance-&gt;lcychdb_addr[unitno] = temp_unit_addr[unitno];                 /* copy the LCTCHDB pattern into lcychdb_loaded in instance_info */                 for (wordno = 0; wordno &lt; 3; ++wordno) {                     unit_ptrn[unitno].word[wordno] =                         preamble_unit_ptrn[unitno].word[wordno];                 }             }             if (&amp;preamble-&gt;word[i] == &amp;preamble-&gt;lcychdb) {                 unit_ptrn = (PTRN_BITS_LONGWORD *)instance-&gt;lcychdb_loaded;                 preamble_unit_ptrn =                     (PTRN_BITS_LONGWORD *)preamble-&gt;word[i];                 temp_unit_addr = instance-&gt;                     unit_addr(instance-&gt;cur_unit_addr_index)[0];                 for (unitno = 0; unitno &lt; dab_ptr-&gt;unit_count; ++unitno) {                     /* copy the addr of LCTCHDB into lcychdb_addr in instance_info */                     instance-&gt;lcychdb_addr[unitno] = temp_unit_addr[unitno];                     /* copy the LCTCHDB pattern into lcychdb_loaded in instance_info */                     for (wordno = 0; wordno &lt; 3; ++wordno) {                         unit_ptrn[unitno].word[wordno] =                             preamble_unit_ptrn[unitno].word[wordno];                     }                 }             }             DPRINTF(("reloading preamble word: %d", i));             write_pattern(instance,                 instance-&gt;unit_addr(instance-&gt;cur_unit_addr_index)[0],                 preamble-&gt;word[i]);             if (grow_pattern(instance) == FAILURE)                 return(FAILURE);         }     }     return(SUCCESS); }  load_pre_feedback_seq(instance) INSTANCE_INFO *instance; {     DEVICE_SPEC *def;     SEQ_SPEC *seq_spec_ptr;     WMB_OFFSET *wmb_ptr;     char *bit_array;     u_short pre_seq_length;     u_short seq_count;     u_short i;     u_short pin_count;     u_short word_offset;     u_short bit_offset;     u_short pin_number;     u_char unitno;     u_char wordno;     u_char bitno; </pre>			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89  
TIME 6:14:43 pm

PAGE #  
8/80

```

LINE # SOURCE TEXT
841
842 DPRINTF(("inside load_pre_feedback_seq\n"));
843
844 def = instance->definition;
845
846 pre_seq_length = def->pre_seq_len;
847 seq_count = def->seq_cnt;
848
849 for (i = 0; i < pre_seq_length; ++i) {
850
851     word_offset = i / 8;
852     bit_offset = 7 - i % 8;
853
854     for (pin_count = 0; pin_count < seq_count; ++pin_count) {
855
856         seq_spec_ptr = def->seq_table[pin_count];
857         bit_array = seq_spec_ptr->pre_bits;
858         pin_number = seq_spec_ptr->pin_number;
859
860         word_ptr = &pre_data[word_offset];
861         unitno = word_ptr->unitno;
862         wordno = word_ptr->wordno;
863         bitno = word_ptr->bitno;
864
865         if ((bit_array[word_offset] >> bit_offset) & 1) {
866             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
867         }
868         else {
869             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
870         }
871     }
872
873     write_patterns(instance,
874                   instance->unit_addr(instance->cur_unit_addr_index)[0],
875                   instance->ptrn_loaded);
876     /* Don't grow the pattern on the last pre feedback sequence pattern.
877     * because otherwise grow_patterns() might allocate new blocks which
878     * will have to be retracted if the pre feedback is followed by
879     * a feedback sequence.
880     */
881     if (i != (pre_seq_length - 1))
882         if (grow_patterns(instance) == FAILURE)
883             return(FAILURE);
884 }
885 return(SUCCESS);
886 }
887
888 load_feedback_seq(instance, overflowed_patterns_count)
889 INSTANCE_INFO *instance;
890 u_long *overflowed_patterns_count;
891 {
892     DEVICE_SPEC def;
893     PTRN_BITS_LONGWORD dummy_long;
894     SEQ_SPEC seq_spec_ptr;
895     DAB_INFO dab_ptr;
896     WORD_OFFSET word_ptr;
897     char *bit_array;
898     u_long dest_addr;
899     u_long source_addr[MAX_LANE_COUNT];
900     u_long saved_block_addr[MAX_LANE_COUNT];
901     u_long temp_unit_addr;
902     short leftover;
903     u_short quiet_patterns;
904     short added_patterns;
905     short extra_patterns;
906     u_char branch_top;
907     u_short dummy_count;
908     short branch_delay;
909     short branch_location;
910     short physical_seq_len;
911     short sub_sequence;
912     u_short pin_number;
913     u_short fb_ptrn_count;
914     short number_of_copies;
915     short bits_per_pin = 1;
916     u_short i;
917     u_short j;
918     u_short played;
919     u_short ptrn_to_copy;
920     short addr_offset;
921     u_short word_offset;
922     u_short pin_count;
923     u_short seq_count;
924     u_short fb_seq_length;
925     u_short power_of_2_ceiling;
926     short dummy_count1;
927     short dummy_count2;
928     u_char total_unit;
929     u_char unitno;
930     u_char wordno;
931     u_char bitno;
932     u_char bit_offset;
933     u_char lane;
934     u_char max_block_size;
935
936 DPRINTF(("inside load_feedback\n"));
937
938 def = instance->definition;
939
940 dab_ptr = dab_list(instance->dab_info_index);
941
942 fb_ptrn_count = 0;
943 if (bits_per_pin == 1) {
944     if ((def->fb_seq_len < dab_ptr->unit_count_per_lane) &
945         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_1_BIT_PER_PIN)) {
946         is_queue_message(ERROR_MSG, "feedback sequence too long, max length: %d",
947                         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_1_BIT_PER_PIN) /
948                         dab_ptr->unit_count_per_lane);
949         return(FAILURE);
950     }
951 }
952 else {
953     if ((def->fb_seq_len < dab_ptr->unit_count_per_lane * 2) &
954         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_2_BIT_PER_PIN)) {
955         is_queue_message(ERROR_MSG, "feedback sequence too long, max length: %d",
956                         (FB_PTRN_COUNT - MAX_DUMMY_COUNT_2_BIT_PER_PIN) /
957                         (dab_ptr->unit_count_per_lane * 2));
958         return(FAILURE);
959     }
960 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89  
TIME 6:14:43 pm

PAGE #  
9/81

```

LINE # SOURCE TEXT
961
962 fb_seq_length = dcb->fb_seq_len;
963 seq_count = dcb->seq_cnt;
964
965 for (i = 0; i < fb_seq_length; ++i) {
966     word_offset = i / 8;
967     bit_offset = 7 - i % 8;
968
969     for (pin_count = 0; pin_count < seq_count; ++pin_count) {
970         seq_spec_ptr = dcb->seq_table[pin_count];
971         bit_array = seq_spec_ptr->fb_bits;
972         pin_number = seq_spec_ptr->pin_number;
973
974         uwb_ptr = dcb->uwb_ptr;
975         unitno = uwb_ptr->unitno;
976         wordno = uwb_ptr->wordno;
977         bitno = uwb_ptr->bitno;
978
979         if ((bit_array[word_offset] >> bit_offset) & 1) {
980             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
981         }
982         else {
983             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
984         }
985     }
986
987     fb_ptrn_count += dcb_ptr->unit_count_per_lane;
988     write_ptrn(instance,
989                instance->unit_addr(instance->cur_unit_addr_index)[0],
990                instance->ptrn_loaded);
991     if (grow_ptrn(instance) == FAILURE)
992         return(FAILURE);
993 }
994
995 physical_seq_len = fb_ptrn_count;
996
997 /* Up to this point the pattern address has been incremented by
998 * unit_count_per_lane units worth of addresses. From now on we
999 * are going to start writing dummy patterns which is only 1 unit
1000 * long with write_ptrn unit(). The write_ptrn unit() will
1001 * look at the LANE_ADDR_INFO.last_unit_addr for the address to
1002 * write to. Since the last call to grow_ptrn has incremented
1003 * the address for unit_count_per_lane units, we need to adjust
1004 * it by calling the adjust_ptrn_addr().
1005 */
1006 adjust_ptrn_addr(instance);
1007
1008 /* We need to wait the following number of PCLK:
1009 * 4 PCLK for the propagation delay from DUT output
1010 * to the input of the synchronizer register.
1011 * Later: this should depend on the frequency
1012 * we are using.
1013 * 2 logical clock since we assume that the DUT output changes
1014 * within 2 logical clock after the input.
1015 * Later: make this delay specifiable by the
1016 * user in the DADDEF.
1017 * 1 logical clock since the clock edges appear for the whole
1018 * logical clock after the pattern is latched
1019 * into the MAGIC register.
1020 */
1021 quiet_patterns = 4 + 3 * (dcb_ptr->unit_count_per_lane * bits_per_pin);
1022
1023 /* The clock edges can only span 3 PCLK maximum. So if
1024 * (unit_count_per_lane * bits_per_pin) is greater than 3 then we can
1025 * actually reduce the quiet_patterns count by the remainder of PCLK where
1026 * there are no edges.
1027 */
1028 extra_patterns = dcb_ptr->unit_count_per_lane * bits_per_pin - 3;
1029 if (extra_patterns < 0)
1030     extra_patterns = 0;
1031
1032 added_patterns = quiet_patterns - extra_patterns;
1033 if (added_patterns < 0)
1034     added_patterns = 0;
1035
1036 /* Add 1 more pattern to make the subsequence pattern even, so that
1037 * the branch command will stay at the odd address.
1038 */
1039 if (even(physical_seq_len + added_patterns))
1040     branch_nop = 0;
1041 else
1042     branch_nop = 1;
1043
1044 dummy_count = added_patterns + branch_nop;
1045 DPRINTF(("FB dummy_count: %d\n", dummy_count));
1046
1047 for (i = 0; i < dummy_count; ++i) {
1048     ++fb_ptrn_count;
1049     write_ptrn_unit(instance, dcb_ptr->dummy_ptrn);
1050     if (grow_ptrn(instance) == FAILURE)
1051         return(FAILURE);
1052 }
1053
1054 sub_sequence = phys_seq_len + added_patterns + branch_nop;
1055 branch_delay = physical_seq_len - extra_patterns +
1056               quiet_patterns + FB_PIPE_DELAY;
1057
1058 if (!even(branch_delay))
1059     ++branch_delay;
1060
1061 branch_location = branch_delay % sub_sequence - 1;
1062
1063 if (branch_location < 0) {
1064     if (branch_location == -1) {
1065         branch_location = sub_sequence - 1;
1066     }
1067     else {
1068         log_queue_message(ERROR_MSG, "internal error: branch location is negative");
1069         return(FAILURE);
1070     }
1071 }
1072
1073 DPRINTF(("FB branch_location: %d\n", branch_location));
1074 set_feedback_branch(instance, branch_location);
1075
1076
1077
1078
1079
1080

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE	5/23/89	PAGE #
TIME	6:14:43 pm	10/82

```

1081  /* Find the next power of 2 number after sub_sequence */
1082  power_of_2_ceiling = 1;
1083  while (power_of_2_ceiling < sub_sequence) {
1084    power_of_2_ceiling <<= 1;
1085  }
1086
1087  dummy_count = 0;
1088
1089  if (power_of_2_ceiling != sub_sequence) {
1090    /* sub_sequence is not power of 2 long --> add more dummies:
1091     * 1. to make it power of 2 long
1092     * 2. to satisfy the BRANCH_LATENCY in the sub_sequence.
1093     * Choose either 1 or 2 whichever produces less dummy ptrns.
1094     */
1095    dummy_count1 = power_of_2_ceiling - sub_sequence;
1096
1097    /* Note the following:
1098     * - branch_location is always odd
1099     * - BRANCH_LATENCY is even
1100     * - sub_sequence is even at this point
1101     * therefore dummy_count2 will always be even
1102     */
1103    dummy_count2 = (branch_location + 1) + BRANCH_LATENCY - sub_sequence;
1104    if (dummy_count2 < 0)
1105      dummy_count2 = 0;
1106
1107    if (dummy_count1 < dummy_count2)
1108      dummy_count = dummy_count1;
1109    else
1110      dummy_count = dummy_count2;
1111
1112    DPRINTF(("FB more dummy_count to satisfy branch constraint: %d\n",
1113      dummy_count));
1114
1115    for (i = 0; i < dummy_count; ++i) {
1116      ++fb_ptrn_count;
1117      write_pattern_unit(instance, dab_ptr->dummy_ptrn);
1118      if (grow_pattern_unit(instance) == FAILURE)
1119        return(FAILURE);
1120    }
1121  }
1122
1123  /* Replicate the feedback patterns to fill the whole feedback blocks */
1124  for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
1125
1126    if (dab_ptr->lane_used[lane0]) {
1127      dest_addr = instance->fb_block_addr[lane0] +
1128        fb_ptrn_count * PTRN_ADDR_INC;
1129
1130      /* Fill the first 512K patterns */
1131      number_of_copies = FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK /
1132        fb_ptrn_count - 1;
1133      DPRINTF(("copy FB sequence %d times\n", number_of_copies));
1134      while (number_of_copies > 0) {
1135        copy_pattern(instance->fb_block_addr[lane0],
1136          dest_addr, fb_ptrn_count);
1137        dest_addr += fb_ptrn_count * PTRN_ADDR_INC;
1138        --number_of_copies;
1139      }
1140
1141      /* Fill the remaining blocks with dummy patterns */
1142      leftover = FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK % fb_ptrn_count;
1143      DPRINTF(("FB fill block with dummy ptrns to 512 ptrns; count: %d\n",
1144        leftover));
1145
1146      dummy_long = (PTRN_BITS_LONGWORD * dab_ptr->dummy_ptrn[lane0]);
1147      while (leftover > 0) {
1148        write_loc_long((u_long *)dest_addr,
1149          dummy_long->word(0));
1150        write_loc_long((u_long *)dest_addr + LANE_SEGMENT_B_OFFSET,
1151          dummy_long->word(1));
1152        write_loc_long((u_long *)dest_addr + LANE_SEGMENT_C_OFFSET,
1153          dummy_long->word(2));
1154
1155        DPRINTF(("FB dummy 408x: 408x 408x 408x\n", dest_addr,
1156          dummy_long->word(0),
1157          dummy_long->word(1),
1158          dummy_long->word(2)));
1159
1160        --leftover;
1161        dest_addr += PTRN_ADDR_INC;
1162      }
1163
1164      /* Replicate the first 512 patterns if necessary */
1165      number_of_copies =
1166        instance->fb_block_size[lane0] / FB128K_FB_BLOCK_COUNT - 1;
1167      DPRINTF(("FB replicate first 512 patterns for %d times\n",
1168        number_of_copies));
1169      while (number_of_copies > 0) {
1170        copy_pattern(instance->fb_block_addr[lane0], dest_addr,
1171          FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK);
1172        dest_addr += FB128K_FB_BLOCK_COUNT * PTRN_PER_BLOCK *
1173          PTRN_ADDR_INC;
1174        --number_of_copies;
1175      }
1176
1177      /* Increment the pattern count to the maximum feedback block. Note
1178       * that this number is greater than the actual pattern memory used
1179       * if the number of feedback block required on each lane is different.
1180       */
1181      max_block_size = 0;
1182      for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
1183        if (instance->fb_block_size[lane0] > max_block_size)
1184          max_block_size = instance->fb_block_size[lane0];
1185      }
1186      instance->pattern_count += (max_block_size * PTRN_PER_BLOCK) -
1187        fb_ptrn_count;
1188      if (allocate_initial_block(instance) == FAILURE)
1189        return(FAILURE);
1190
1191      /* subtract pattern count which was incremented in
1192       * allocate_initial_block().
1193       */
1194      instance->pattern_count -= dab_ptr->unit_count_per_lane;
1195    }
1196  }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/initseq.c	DATE 5/23/89	PAGE # 11/83
LINE #		SOURCE TEXT		
1201		/* Set the branch address for each block in the feedback sequence to		
1202		* link the post feedback blocks to the feedback blocks.		
1203		*/		
1204		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
1205		if (dab_ptr->lane_used[lane]) {		
1206		for (j = 0; j < instance->fb_block_count[lane]; ++j) {		
1207		write_branch_table(instance->fb_block_addr[lane] +		
1208		j * BLOCK_ADDR_INC,		
1209		instance->lane_addr[lane].max_addr -		
1210		BLOCK_ADDR_INC);		
1211		}		
1212		}		
1213		}		
1214		/* Now copy the part of the feedback sequence to the post feedback blocks		
1215		* because we might have played only part of the feedback sequence due		
1216		* to the BRANCH LATENCY.		
1217		*/		
1218		/* branch_location + 1 -> to get count instead of pattern number */		
1219		played = (branch_location + 1 + BRANCH_LATENCY) %		
1220		(sub_sequence + dummy_count);		
1221		ptrs_to_copy = 0;		
1222		if (physical_seq_len > played)		
1223		ptrs_to_copy = physical_seq_len - played;		
1224		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
1225		if (dab_ptr->lane_used[lane]) {		
1226		source_addr[lane] = instance->fb_block_addr[lane] +		
1227		played * PTRN_ADDR_INC;		
1228		}		
1229		}		
1230		instance->pattern_count += ptrs_to_copy;		
1231		PRINTF("copy fb sequence to post feedback blocks, ptrs_to_copy: %d\n",		
1232		ptrs_to_copy);		
1233		/* ptrs_to_copy can be up to 512 patterns, so we cannot		
1234		* just do a copy_pattern() because we have only allocated 1 block.		
1235		*/		
1236		if (ptrs_to_copy > PTRN_PER_BLOCK) {		
1237		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
1238		if (dab_ptr->lane_used[lane]) {		
1239		copy_pattern(source_addr[lane],		
1240		instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC,		
1241		PTRN_PER_BLOCK);		
1242		source_addr[lane] += BLOCK_ADDR_INC;		
1243		saved_block_addr[lane] = instance->lane_addr[lane].max_addr -		
1244		BLOCK_ADDR_INC;		
1245		}		
1246		}		
1247		ptrs_to_copy -= PTRN_PER_BLOCK;		
1248		if (allocate_initial_block(instance) == FAILURE)		
1249		return(FAILURE);		
1250		/* subtract pattern count which was incremented in		
1251		* allocate_initial_block().		
1252		*/		
1253		instance->pattern_count -= dab_ptr->unit_count_per_lane;		
1254		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
1255		if (dab_ptr->lane_used[lane]) {		
1256		set_branch(saved_block_addr[lane] + BLOCK_ADDR_INC -		
1257		PTRN_ADDR_INC,		
1258		instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC);		
1259		}		
1260		}		
1261		if (ptrs_to_copy > 0) {		
1262		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
1263		if (dab_ptr->lane_used[lane]) {		
1264		copy_pattern(source_addr[lane],		
1265		instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC,		
1266		ptrs_to_copy);		
1267		}		
1268		}		
1269		/*overflowed_pattern_count = ptrs_to_copy;		
1270		/* Setup the pattern unit addresses.		
1271		* make instance->unit_addr[] points to the last pattern which has		
1272		* already been copied by copy_pattern(). When we do this it is possible		
1273		* that some instance->unit_addr[] will point to addresses less than the		
1274		* beginning address of the block. But this is OK because we will call		
1275		* grow_pattern() next which will fix this problem. grow_pattern() will		
1276		* make instance->unit_addr[] points to the address to load the next		
1277		* pattern.		
1278		*/		
1279		/* Calculate the addr offset for each unit addr. Note that we can pick		
1280		* any unit in any lane to calculate the offset, since this offset is		
1281		* the same for any unit. Also note that the addr_offset could be negative		
1282		* if the ptrs_to_copy is less than unit_count_per_lane.		
1283		*/		
1284		lane = dab_ptr->unit_location(0).lane_no;		
1285		temp_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0];		
1286		addr_offset = instance->lane_addr[lane].max_addr - BLOCK_ADDR_INC +		
1287		ptrs_to_copy * PTRN_ADDR_INC -		
1288		dab_ptr->unit_count_per_lane * PTRN_ADDR_INC -		
1289		temp_unit_addr[0];		
1290		total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;		
1291		for (unitno = 0; unitno < total_unit; ++unitno) {		
1292		temp_unit_addr[unitno] += addr_offset;		
1293		}		
1294		if (grow_pattern(instance) == FAILURE)		
1295		return(FAILURE);		
1296		return(SUCCESS);		
1297		}		
1298		}		
1299		}		
1300		}		
1301		}		
1302		}		
1303		}		
1304		}		
1305		}		
1306		}		
1307		}		
1308		}		
1309		}		
1310		}		
1311		}		
1312		}		
1313		}		
1314		}		
1315		}		
1316		}		
1317		}		
1318		}		
1319		}		
1320		}		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89 PAGE #  
TIME 6:14:43 pm 12/84

```

1321
1322 load_post_feedback_seq(instance)
1323 INSTANCE_INFO *instance;
1324 {
1325     DEVICE_SPEC *def;
1326     SEQ_SPEC *seq_spec_ptr;
1327     UWB_OFFSET *uwb_ptr;
1328     char *bit_array;
1329     int i;
1330     u_short pin_count;
1331     u_short word_offset;
1332     u_short bit_offset;
1333     u_short pin_reset;
1334     u_short post_seq_length;
1335     u_short seq_count;
1336     u_char unitno;
1337     u_char wordno;
1338     u_char bitno;
1339
1340     DPRINTF(("inside load_post_feedback_seq\n"));
1341
1342     def = instance->definition;
1343     seq_count = def->seq_cnt;
1344     post_seq_length = def->post_seq_len;
1345
1346     for (i = 0; i < post_seq_length; ++i) {
1347
1348         word_offset = i / 8;
1349         bit_offset = 7 - i % 8;
1350
1351         for (pin_count = 0; pin_count < seq_count; ++pin_count) {
1352
1353             seq_spec_ptr = def->seq_table[pin_count];
1354             bit_array = seq_spec_ptr->post_bits;
1355             pin_number = seq_spec_ptr->pin_number;
1356
1357             uwb_ptr = uwb_to_short_offset(pin_number);
1358             unitno = uwb_ptr->unitno;
1359             wordno = uwb_ptr->wordno;
1360             bitno = uwb_ptr->bitno;
1361
1362             if ((bit_array[word_offset] >> bit_offset) & 1) {
1363                 set_ptrn_bit(instance->ptrn_loaded, wordno, bitno);
1364             }
1365             else {
1366                 reset_ptrn_bit(instance->ptrn_loaded, wordno, bitno);
1367             }
1368         }
1369
1370         write_pattern(instance,
1371                     instance->unit_addr[instance->cur_unit_addr_index][0],
1372                     instance->ptrn_loaded);
1373         if (grow_pattern(instance) == FAILURE)
1374             return(FAILURE);
1375     }
1376     return(SUCCESS);
1377
1378
1379 set_initial_values(instance)
1380 INSTANCE_INFO *instance;
1381 {
1382     DEVICE_SPEC *def_ptr;
1383     EXTRA_DEVICE_SPEC *extra_def_ptr;
1384     DAB_INFO *dab_ptr;
1385     PTRN_SPEC *ptrn_spec_ptr;
1386     PIN_INFO *pin_info;
1387     UWB_OFFSET *uwb_ptr;
1388     PTRN_BITS *ptrn_bits_ptr;
1389     PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
1390     PTRN_BITS_LONGWORD *last_consistent_set_ptr;
1391     PTRN_BITS_LONGWORD *ident_loc_ptr;
1392     PTRN_BITS_LONGWORD *ident_store_ptr;
1393     PTRN_BITS_LONGWORD *ident_outputs_ptr;
1394     PTRN_BITS_LONGWORD *ident_R1_ptr;
1395     PTRN_BITS_LONGWORD *ident_R2_ptr;
1396     PTRN_BITS_LONGWORD *hndcnb_loaded_ptr;
1397     PTRN_BITS_LONGWORD *last_sample_value_hiz_ptr;
1398     u_long last_block_number[MAX_LANE_COUNT];
1399     u_long ident_store_loc;
1400     u_long seq_end_addr[MAX_LANE_COUNT];
1401     u_short seq_count;
1402     u_short pin_count;
1403     u_short pin_number;
1404     u_char total_unit;
1405     u_char unitno;
1406     u_char wordno;
1407     u_char bitno;
1408     u_char changed_dac;
1409
1410     DPRINTF(("set initial values\n"));
1411
1412     def_ptr = instance->definition;
1413     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
1414
1415     dab_ptr = dab_list(instance->dab_info_index);
1416     total_unit = dab_ptr->unit_count;
1417
1418     ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
1419     ident_R1_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R1;
1420     ident_R2_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R2;
1421     ident_outputs_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_outputs;
1422     for (unitno = 0; unitno < total_unit; ++unitno) {
1423         for (wordno = 0; wordno < 3; ++wordno) {
1424
1425             /* Disable the R1 and R2 Clock */
1426             ptrn_loaded_ptr->word[wordno] |= ident_R1_ptr->word[wordno];
1427             ptrn_loaded_ptr->word[wordno] |= ident_R2_ptr->word[wordno];
1428
1429             ptrn_loaded_ptr->word[wordno] |= ident_outputs_ptr->word[wordno];
1430         }
1431
1432         ++ptrn_loaded_ptr;
1433         ++ident_R1_ptr;
1434         ++ident_R2_ptr;
1435         ++ident_outputs_ptr;
1436     }
1437
1438     write_pattern(instance,
1439                 instance->unit_addr[instance->cur_unit_addr_index][0],
1440                 instance->ptrn_loaded);

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/initseq.c	DATE 5/23/89	PAGE # 13/85
LINE #	SOURCE TEXT			
1441	/*ifdef DBASE			
1442	if (start_seq(def_ptr) == FAILURE) {			
1443	return(FAILURE);			
1444	}			
1445	endif			
1446	if (program_dac(def_ptr, (char)instance->dab_info_index,			
1447	changed_dac) == FAILURE) {			
1448	return(FAILURE);			
1449	}			
1450	if (set_edge_and_sample_setting(extra_def_ptr,			
1451	(u_long)RESOLUTION_05_NS,			
1452	(u_long)MAX_SAMPLE_RANGE) == FAILURE)			
1453	return(FAILURE);			
1454	set_seq_end_bit(instance,			
1455	instance->last_addr,			
1456	FALSE,			
1457	seq_end_addr,			
1458	inst_block_number);			
1459	if (play_ptr_seq(instance, (u_long)PTEN_PLAY_TIMEOUT,			
1460	changed_dac) == FAILURE)			
1461	return(FAILURE);			
1462	/*ifdef DEBUG			
1463	if (loop_till_key == TRUE) {			
1464	printf("looping in create instance\n");			
1465	debug_key = 0;			
1466	while (debug_key == 0) {			
1467	if (play_ptr_seq(instance,			
1468	(u_long)PTEN_PLAY_TIMEOUT, changed_dac) == FAILURE)			
1469	return(FAILURE);			
1470	}			
1471	if (debug_key == (u_long)'i') {			
1472	loop_till_key = FALSE;			
1473	printf("stop loop\n");			
1474	}			
1475	debug_key = 0;			
1476	endif			
1477	remove_seq_end_bit(instance, seq_end_addr, inst_block_number);			
1478	read_magic_full_sample_reg(instance, &instance->last_sample_value);			
1479	/* Set the initial simulator pin values to whatever value we sample. */			
1480	copy_ptr_bits((char *)instance->last_sample_value.unknown,			
1481	(char *)instance->sim_pin_value.unknown,			
1482	dab_ptr->unit_count);			
1483	copy_ptr_bits((char *)instance->last_sample_value.hiz,			
1484	(char *)instance->sim_pin_value.hiz,			
1485	dab_ptr->unit_count);			
1486	copy_ptr_bits((char *)instance->last_sample_value.data,			
1487	(char *)instance->sim_pin_value.data,			
1488	dab_ptr->unit_count);			
1489	/* Set the sim_pin_value to the last value of the reset sequence.			
1490	* Note that this value is the last ptrn loaded.			
1491	* Also note that for RL/RS pins, set the sim_pin_value to the inactive			
1492	* state.			
1493	*/			
1494	seq_count = def_ptr->seq_cnt;			
1495	for (pin_count = 0; pin_count < seq_count; ++pin_count) {			
1496	pin_number = def_ptr->seq_table[pin_count].pin_number;			
1497	pin_spec_ptr = &def_ptr->pin_table[pin_number];			
1498	pin_info = &instance->pin_info_table[pin_number];			
1499	word_ptr = &pin_info->word_ptr;			
1500	unitno = word_ptr->unitno;			
1501	wordno = word_ptr->wordno;			
1502	bitno = word_ptr->bitno;			
1503	switch (pin_spec_ptr->clk_format) {			
1504	case DWR1:			
1505	pin_info->uninitialized_pin = FALSE;			
1506	if (read_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno) == 0)			
1507	set_pin_value(instance->sim_pin_value,			
1508	unitno, wordno, bitno, (u_char)LOGIC_0);			
1509	else			
1510	set_pin_value(instance->sim_pin_value,			
1511	unitno, wordno, bitno, (u_char)LOGIC_1);			
1512	break;			
1513	case R1:			
1514	pin_info->uninitialized_pin = FALSE;			
1515	set_pin_value(instance->sim_pin_value,			
1516	unitno, wordno, bitno, (u_char)LOGIC_1);			
1517	break;			
1518	case R0:			
1519	pin_info->uninitialized_pin = FALSE;			
1520	set_pin_value(instance->sim_pin_value,			
1521	unitno, wordno, bitno, (u_char)LOGIC_0);			
1522	break;			
1523	default:			
1524	break;			
1525	}			
1526	/* Set the initial simulator pin value for I/O store pins to 1.			
1527	* Also set the RNDENB for I/O store pins which are driving to 1.			
1528	*/			
1529	ident_ios_ptr = (PTEN_BITS_LONGWORD *)extra_def_ptr->ident_ios;			
1530	ident_store_ptr = (PTEN_BITS_LONGWORD *)extra_def_ptr->ident_store;			
1531	hndenb_loaded_ptr = (PTEN_BITS_LONGWORD *)instance->hndenb_loaded;			
1532	last_sample_value_hiz_ptr =			
1533	(PTEN_BITS_LONGWORD *)instance->last_sample_value.hiz;			
1534	for (unitno = 0; unitno < total_unit; ++unitno) {			
1535	for (wordno = 0; wordno < 1; ++wordno) {			
1536	ident_store_ios = ident_ios_ptr->word[wordno] &			
1537	ident_store_ptr->word[wordno];			
1538	hndenb_loaded_ptr->word[wordno]  =			
1539	ident_store_ios & last_sample_value_hiz_ptr->word[wordno];			
1540	}			
1541	++ident_ios_ptr;			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/initseq.c

DATE 5/23/89  
TIME 6:14:43 pm

PAGE #  
14/86

```

1561  ++ident_store_ptr,
1562  ++hmdenb_loaded_ptr,
1563  ++last_sample_value_ptr,
1564  }
1565
1566  /* Calculate consistent set */
1567  calculate_consistent_set(instance, def_ptr, instance->last_consistent_set);
1568
1569  /* Restore the HMDENB to the actual value. The first HMDENB (in preamble)
1570   * disable the I/O driver for IO and OUTPUT pins.
1571   */
1572  write_pattern(instance,
1573               instance->unit_addr[instance->cur_unit_addr_index][0],
1574               instance->hmdenb_loaded);
1575
1576  if (grow_pattern(instance) == FAILURE)
1577      return(FAILURE);
1578
1579  last_consistent_set_ptr = (PTRN_BITS_LONGWORD *)
1580                          instance->last_consistent_set;
1581  ptrn_bits_ptr = instance->last_consistent_set;
1582  ident_R1_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R1;
1583  ident_R2_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_R2;
1584  for (unitno = 0; unitno < total_unit; ++unitno) {
1585      set_pel_user_bit(ptrn_bits_ptr->ctl);
1586
1587      for (wordno = 0; wordno < 3; ++wordno) {
1588
1589          /* Disable the R1 and R2 clock */
1590          last_consistent_set_ptr->word[wordno] |= ident_R1_ptr->word[wordno];
1591          last_consistent_set_ptr->word[wordno] |= ident_R2_ptr->word[wordno];
1592      }
1593
1594      ++last_consistent_set_ptr,
1595      ++ident_R1_ptr,
1596      ++ident_R2_ptr,
1597      ++ptrn_bits_ptr,
1598  }
1599
1600  /* Mail down the first pattern */
1601  write_pattern(instance,
1602               instance->unit_addr[instance->cur_unit_addr_index][0],
1603               instance->last_consistent_set);
1604
1605  /* User bit in first pattern only. */
1606  ptrn_bits_ptr = instance->last_consistent_set;
1607  for (unitno = 0; unitno < total_unit; ++unitno) {
1608      clear_pel_user_bit(ptrn_bits_ptr->ctl);
1609      ++ptrn_bits_ptr;
1610  }
1611
1612  if (grow_pattern(instance) == FAILURE)
1613      return(FAILURE);
1614
1615  /* Duplicate the first pattern (with the trigger bit) because,
1616   * it can be overwritten by the HMDENB if any I/O store pins
1617   * change direction from driving to Z.
1618   */
1619  write_pattern(instance,
1620               instance->unit_addr[instance->cur_unit_addr_index][0],
1621               instance->last_consistent_set);
1622
1623  if (grow_pattern(instance) == FAILURE)
1624      return(FAILURE);
1625
1626  /* copy the last_consistent_set to ptrn_loaded */
1627  copy_ptrn_bits((char *)instance->last_consistent_set,
1628               (char *)instance->ptrn_loaded,
1629               dab_ptr->unit_count);
1630
1631  /* Initialize the OUTPUT pins in ptrn_loaded to 0 */
1632  ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
1633  ident_outputs_ptr = (PTRN_BITS_LONGWORD *)extra_def_ptr->ident_outputs;
1634  for (unitno = 0; unitno < total_unit; ++unitno) {
1635      for (wordno = 0; wordno < 3; ++wordno) {
1636          ptrn_loaded_ptr->word[wordno] |= ident_outputs_ptr->word[wordno];
1637      }
1638      ++ptrn_loaded_ptr,
1639      ++ident_outputs_ptr,
1640  }
1641
1642  /* Write out the measurement pattern */
1643  write_pattern(instance,
1644               instance->unit_addr[instance->cur_unit_addr_index][0],
1645               instance->ptrn_loaded);
1646
1647  return(SUCCESS);
1648
1649  }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/lmserver.c	DATE 5/23/89	PAGE # 1/87
LINE #	SOURCE TEXT			
1	/* SCCS ID: lmserver.c rev 3.2, 5/9/89 at 17:20:35 */			
2	#include "device.h"			
3	#include "message.h"			
4	#include "re.h"			
5	#include "soc_x.h"			
6	#include "septom.h"			
7	#include "lmserver.h"			
8	#include "network.h"			
9	#include "function.h"			
10	#include "lm_sfi.h"			
11	#endif			
12	#ifdef MODELER			
13	#include "vtx.h"			
14	#endif			
15	#ifdef DEBUG			
16	extern u_long debug_key;			
17	extern u_long debug_char;			
18	u_char loop_till_key = FALSE;			
19	#endif			
20	/* ??? defined in wvarm.h */			
21	#define SHUTDOWN (char)2			
22	#define REBOOT (char)3			
23	#define TICKS_PER_SECOND 1000			
24	#define REBOOT_MSG_DELAY_TIME (TICKS_PER_SECOND * 30)			
25	#ifdef DIRECTCONN			
26	ec_spend(X, Y, Z) {}			
27	#endif			
28	u_char play_completed_flag;			
29	u_long available_malloc_size;			
30	u_long total_malloc_size;			
31	int play_semaphore;			
32	#endif			
33	#ifdef DEBUG			
34	u_char lm_printit = TRUE;			
35	#endif			
36	CONNECTION *lm_global_conn_ptr;			
37	extern CONNECTION table_of_conns[1];			
38	#endif			
39	DAB_INFO *dab_list[MAX_LANE_COUNT * MAX_SLOT_COUNT];			
40	USER_INFO *user_info_array[MAX_USER_COUNT];			
41	SYSTEM_INFO *system_config;			
42	#endif			
43	void (*function_array[MAX_FUNCTION])();			
44	#endif			
45	CONFIGURATION_ERRORS config_error;			
46	/* CPU byte address of free block on each lane */			
47	u_long free_block_list[MAX_LANE_COUNT];			
48	#endif			
49	u_long bitso_to_mask[32];			
50	#endif			
51	/* ??? for debugging only */			
52	#ifdef DBASE			
53	TM_RESULT tm_result_array[MAX_PIN_COUNT];			
54	#endif			
55	u_char is_measure_delay;			
56	#endif			
57	PTRN_BITS *gbl_dummy_ptrn;			
58	#endif			
59	FULL_VALUE gbl_new_sample_value;			
60	FULL_VALUE gbl_steady_state_result;			
61	FULL_VALUE gbl_temp_steady_state_result;			
62	PTRN_BITS_LONGWORD gbl_idest_inconsistent_ptrn[MAX_UNIT_COUNT];			
63	PTRN_BITS_LONGWORD gbl_idest_change[MAX_UNIT_COUNT];			
64	TM_PIN_INFO gbl_tm_pin_info_table[MAX_PIN_COUNT];			
65	#endif			
66	/* The convert bit patterns to logic value.			
67	/* The syntax of the bit patterns is ONE:012:SOFT:DATA			
68	*/			
69	u_char bit_to_logic_table[16];			
70	#endif			
71	#ifdef MODELER			
72	LM_HARDWARE_ERROR modeler_error;			
73	#endif			
74	extern LM_HARDWARE_ERROR modeler_error;			
75	u_char fatal_hardware_error_encountered = FALSE;			
76	u_char fatal_configuration_error_encountered = FALSE;			
77	u_char non_fatal_configuration_error_encountered = FALSE;			
78	long time_reboot_was_issued;			
79	long reboot_delay_time;			
80	u_char rebooting = FALSE;			
81	u_char reboot_flag;			
82	u_char xabutdown = FALSE;			
83	#endif			
84	UWB_OFFSET ps_to_short_offset[MAX_PIN_COUNT];			
85	#endif			
86	u_short gbl_eval_pin_number[MAX_EVAL_CHANGES];			
87	u_char gbl_eval_pin_value[MAX_EVAL_CHANGES];			
88	u_long gbl_eval_pin_delay[MAX_EVAL_CHANGES];			
89	u_long gbl_eval_max_delay[MAX_EVAL_CHANGES];			
90	#endif			
91	u_char lm_hardware_init_done = FALSE;			
92	#endif			
93	extern u_long lm_dead_user_mask;			
94	#endif			
95	#ifdef MODELER			
96	main()			
97	{			
98	false			
99	void			
100	pattern_task()			
101	#endif			
102	{			
103	u_short rtn;			
104	u_char user;			
105	u_short type;			
106	u_long temp;			
107	u_long minis_secs;			
108	char str[MAX_MESSAGE];			
109	u_long *reboot_msg_delay = (u_long*) NULL;			
110	#endif			
111	#endif			
112	{			
113	u_short rtn;			
114	u_char user;			
115	u_short type;			
116	u_long temp;			
117	u_long minis_secs;			
118	char str[MAX_MESSAGE];			
119	u_long *reboot_msg_delay = (u_long*) NULL;			
120	#endif			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/lmserver.c

DATE 5/23/89  
TIME 6:14:45 pm

PAGE #  
2/88

```

121  extern u_short lm_give_me_the_next_user();
122
123  DPRINTF(("LM server\n"));
124
125  #ifdef MODELER
126  init();
127
128  if (init_socket() == FAILURE) {
129      DPRINTF(("ERROR in init_socket\n"));
130  }
131
132  read_hw_config();
133
134  init_mod_err();
135
136  (void)printf("ready\n");
137  #endif
138
139  lm_dead_user_mask = 0;
140  while (1) {
141      rta = lm_give_me_the_next_user(user);
142      DPRINTF(("got user: id from lm_give_me_the_next_user\n", user));
143
144      lm_global_conn_ptr = table_of_conns[user];
145      if (lm_global_conn_ptr == NULL)
146          continue;
147
148      if (rta != SUCCESS) {
149          while (lm_dequeue_message(&type, &str) == SUCCESS) {
150              DPRINTF(("ta\n", str));
151          }
152          DPRINTF(("error in lm_give_me_the_next_user()\n"));
153      }
154      else {
155          lm_flush_message_queue();
156
157          if (lm_hardware_init_done == FALSE) {
158              send_hardware_init_message();
159              continue;
160          }
161
162          if (user_info_array[user] != active == FALSE) {
163              if (rebooting == FALSE) {
164                  if (prepare_user(user) == FAILURE) {
165                      send_alloc_error_message(user_info_array[user]);
166                      check_reboot();
167                      continue;
168                  }
169              }
170
171              if (non_fatal_configuration_error_encountered == TRUE) {
172                  queue_up_non_fatal_config_message();
173              }
174
175              process_client_request(user_info_array[user]);
176          }
177          else {
178              if (prepare_user(user) == FAILURE) {
179                  send_alloc_error_message(user_info_array[user]);
180                  check_reboot();
181                  continue;
182              }
183
184              send_rebooting_message(user_info_array[user]);
185          }
186      }
187      else {
188          if (rebooting == TRUE) {
189              if (reboot_msg_delay == (u_long *) NULL) {
190                  reboot_msg_delay = (u_long *) DCALLOC(MAX_USER_COUNT, sizeof(u_long));
191              }
192              temp = lm_time() - time_reboot_was_issued;
193              if (temp > (reboot_msg_delay + user)) {
194                  if (reboot_delay_time > temp) {
195                      mins = (reboot_delay_time - temp) / (60 * TICKS_PER_SECOND);
196                      secs = ((reboot_delay_time - temp) / TICKS_PER_SECOND) - (mins * 60);
197                      lm_queue_message(WARNING_MSG, "Modeler is going down in %d minutes and %d seconds.",
198                                     mins, secs);
199                  }
200                  else {
201                      lm_queue_message(WARNING_MSG, "Modeler is going down when all users are finished.");
202                  }
203                  *(reboot_msg_delay + user) = temp + REBOOT_MSG_DELAY_TIME;
204              }
205          }
206          process_client_request(user_info_array[user]);
207      }
208  }
209  }
210
211  check_reboot();
212
213  }
214
215  #endif
216
217  process_client_request(cur_user)
218  USER_INFO *cur_user;
219  {
220      u_short errors;
221      u_short warnings;
222      char func_number;
223
224      DPRINTF(("inside process_client_request\n"));
225
226      #ifdef DEBUG
227      if (debug_key == 1) {
228          printf("key hit: %x\n", debug_char);
229          if (debug_char == (u_long)'p') {
230              if (lm_printit == TRUE)
231                  lm_printit = FALSE;
232              else
233                  lm_printit = TRUE;
234          }
235          else if ((debug_char == (u_long)'t') {
236              if (loop_till_key == TRUE)
237                  loop_till_key = FALSE;
238              else
239                  loop_till_key = TRUE;
240          }
241      }
242  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/lmserver.c

DATE 5/23/89  
TIME 6:14:45 pm

PAGE #  
3/89

```

LINE # SOURCE TEXT
241     else {
242         printf("unknown key hit\n");
243     }
244     debug_key = 0;
245 }
246 #endif
247
248 if (fatal_hardware_error_encountered == TRUE) {
249     DPRINTF(("fatal_hardware_error_encountered == TRUE\n"));
250     send_fatal_hw_error_message(cur_user);
251     return;
252 }
253 else if (fatal_configuration_error_encountered == TRUE) {
254     DPRINTF(("fatal_configuration_error_encountered == TRUE\n"));
255     send_fatal_config_error_message(cur_user);
256     return;
257 }
258 else {
259     if (modeler_error.error == TRUE) {
260         DPRINTF(("modeler_error.error == TRUE\n"));
261         modeler_error.error = FALSE;
262
263         if ((modeler_error.pac_lane_errors != 0) ||
264             (modeler_error.lms_error == TRUE) ||
265             (modeler_error.unknown_source_of_interrupt == TRUE)) {
266             fatal_hardware_error_encountered = TRUE;
267             send_fatal_hw_error_message(cur_user);
268             return;
269         }
270
271         if (modeler_error.dab_change) {
272             reconfigure_dab();
273         }
274         else {
275             if (modeler_error.pel_error_list != 0) {
276                 /* Set it back to TRUE so that play_ptr_seq() is aware
277                  * of this condition.
278                  */
279                 modeler_error.error = TRUE;
280             }
281         }
282     }
283
284     func_number = (lm_get_int() - 2) >> 1;
285
286     if ((func_number >= 0) && (func_number < MAX_FUNCTION)) {
287         function_array[func_number](cur_user);
288     }
289     else
290         process_no_such(cur_user);
291 }
292
293 lm_message_types(&errors, &warnings);
294 if ((errors + warnings) != 0)
295     DPRINTF(("ASSERT: some messages not dequeued at end of process.\n"));
296
297 #endif
298
299 send_hardware_init_message()
300 {
301     char command_number;
302     #ifdef DEBUG
303     u_short temp;
304     char error_string[MAX_STRING_LENGTH];
305 #endif
306     DPRINTF(("inside send_hardware_init_message\n"));
307
308     command_number = lm_get_int();
309
310     lm_put_int(command_number + 1);
311     lm_queue_message(ERROR_MSG, "modeler is being initialized");
312     end_queue_message();
313
314     end_put(0);
315
316     lm_delay(50);
317
318     if (close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
319         #ifdef DEBUG
320         while (1) {
321             if (lm_dequeue_message(&temp, error_string) != FAILURE)
322                 DPRINTF(("%" error_string));
323             else
324                 break;
325         }
326         #endif
327     }
328 }
329
330
331 send_rebooting_message(user)
332 USER_INFO *user;
333 {
334     #ifdef DEBUG
335     u_short temp;
336     char error_string[512];
337     #endif
338     char command_number;
339
340     DPRINTF(("inside send_rebooting_message\n"));
341
342     command_number = lm_get_int();
343
344     reset_obuf();
345     lm_put_int(command_number + 1);
346     if (xshutdown == FALSE) {
347         lm_queue_message(ERROR_MSG, "cannot accept any new users; modeler is being rebooted...");
348     } else {
349         lm_queue_message(ERROR_MSG, "cannot accept any new users; modeler is being shut down...");
350     }
351
352     abort_user(user);
353
354     end_queue_message();
355
356     end_put(user->fd);
357
358     if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
359         #ifdef DEBUG

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/lmsrver.c

DATE 5/23/89  
TIME 6:14:45 pm

PAGE #  
4/90

LINE #	SOURCE TEXT
361	while (1) {
362	if (lm_dequeue_message(&temp, error_string) != FAILURE)
363	DPRINTF((" %a", error_string));
364	else
365	break;
366	}
367	endif
368	}
369	}
370	
371	send_fatal_hw_error_message(user)
372	USER_INFO *user;
373	{
374	#ifdef DEBUG
375	u_short temp;
376	char error_string[512];
377	endif
378	char command_number;
379	
380	DPRINTF(("inside send_fatal_hw_error_message\n"));
381	
382	command_number = lm_get_int();
383	
384	reset_obuf();
385	lm_put_int(command_number + 1);
386	
387	get_fatal_hardware_message();
388	
389	abort_user(user);
390	
391	end_queue_message();
392	
393	end_put(user->fd);
394	
395	
396	if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
397	#ifdef DEBUG
398	while (1) {
399	if (lm_dequeue_message(&temp, error_string) != FAILURE)
400	DPRINTF((" %a", error_string));
401	else
402	break;
403	}
404	endif
405	}
406	
407	
408	send_fatal_config_error_message(user)
409	USER_INFO *user;
410	{
411	u_short temp;
412	char command_number;
413	#ifdef DEBUG
414	char error_string[512];
415	endif
416	
417	DPRINTF(("inside send_fatal_config_error_message\n"));
418	
419	command_number = lm_get_int();
420	
421	reset_obuf();
422	lm_put_int(command_number + 1);
423	
424	lm_queue_message(ERROR_MSG, "Fatal error encountered during hardware configuration");
425	if (config_error.no_tmg) {
426	lm_queue_message(ERROR_MSG, "Timing Generator is offline");
427	}
428	
429	if (config_error.tmg_cal) {
430	lm_queue_message(ERROR_MSG, "Timing Generator error: calibration failed");
431	}
432	
433	if (config_error.no_pam_for_pac) {
434	for (temp = 0; temp < MAX_LANE_COUNT; ++temp) {
435	if (config_error.no_pam_for_pac & (1 << temp)) {
436	lm_queue_message(ERROR_MSG, "No Fast Pattern Memory in lane: %c",
437	'A' + temp);
438	}
439	}
440	
441	
442	if (config_error.pam_strapped_wrong) {
443	for (temp = 0; temp < MAX_LANE_COUNT * MAX_PAM_COUNT; ++temp) {
444	if (config_error.pam_strapped_wrong & (1 << temp)) {
445	lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c is strapped incorrectly",
446	temp % MAX_PAM_COUNT,
447	'A' + temp / MAX_PAM_COUNT);
448	}
449	}
450	
451	
452	if (config_error.pam_stacked_wrong) {
453	for (temp = 0; temp < MAX_LANE_COUNT * MAX_PAM_COUNT; ++temp) {
454	if (config_error.pam_stacked_wrong & (1 << temp)) {
455	lm_queue_message(ERROR_MSG, "Fast Pattern Memory: %d in lane: %c is stacked incorrectly",
456	temp % MAX_PAM_COUNT,
457	'A' + temp / MAX_PAM_COUNT);
458	}
459	}
460	
461	
462	abort_user(user);
463	
464	end_queue_message();
465	
466	end_put(user->fd);
467	
468	
469	if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
470	#ifdef DEBUG
471	while (1) {
472	if (lm_dequeue_message(&temp, error_string) != FAILURE)
473	DPRINTF((" %a", error_string));
474	else
475	break;
476	}
477	endif
478	}
479	
480	

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/lmserver.c

DATE

5/23/89

PAGE #

TIME

6:14:45 pm

5/91

```

LINE # SOURCE TEXT
481 read_alloc_error_message(user)
482 USER_INFO "user",
483 {
484 #ifdef DEBUG
485     u_short temp;
486     char error_string[512];
487 #endif
488     char command_number;
489
490     DPRINTF(("inside read_alloc_error_message\n"));
491
492     command_number = lm_get_int();
493
494     reset_buf();
495     lm_put_int(command_number + 1);
496     lm_queue_message(ERROR_MSG, "out of memory on modeler; user aborted...");
497     abort_user(user);
498
499     end_queue_message();
500
501     end_put(user->fd);
502
503
504
505     if (set_close_connection_for_server(lm_global_conn_ptr) != SUCCESS) {
506 #ifdef DEBUG
507         while (1) {
508             if (lm_dequeue_message(temp, error_string) != FAILURE)
509                 DPRINTF(("ls", error_string));
510             else
511                 break;
512         }
513 #endif
514     }
515
516     check_reboot()
517
518     /* Read any modeler address space to force it to interrupt if there is
519     * any dab configuration changes.
520     */
521
522     (void)probe((u_long)CLOS_START_ADDR);
523
524     if (rebooting == TRUE) {
525         if ((lm_time() - time_reboot_was_issued) > reboot_delay_time) {
526             /* The number of seconds we need to wait has elapsed */
527             if (reboot_flag == LM_FALSE) {
528                 /* We don't need to wait for all users to finish --
529                 * just hit the software reset button now.
530                 */
531                 if (xshutdwn == FALSE) {
532                     #ifdef MODELER
533                         lm_delay(1000);
534                         reset_cpu(REBOOT);
535                     #endif
536                 }
537             }
538             else {
539                 #ifdef MODELER
540                     lm_delay(1000);
541                     reset_cpu(SETDOWN);
542                 #endif
543             }
544         }
545         else {
546             if (any_active_user() == FALSE) {
547                 if (xshutdwn == FALSE) {
548                     #ifdef MODELER
549                         lm_delay(1000);
550                         reset_cpu(REBOOT);
551                     #endif
552                 }
553                 else {
554                     #ifdef MODELER
555                         lm_delay(1000);
556                         reset_cpu(SETDOWN);
557                     #endif
558                 }
559             }
560         }
561     }
562 }
563
564 any_active_user()
565 {
566     u_char i;
567
568     for (i = 0; i < MAX_USER_COUNT; ++i)
569         if (user_info_array[i]->active == TRUE)
570             return(TRUE);
571
572     return(FALSE);
573 }
574
575 queue_up_non_fatal_config_message()
576 {
577     u_char slotno;
578
579     DPRINTF(("inside queue_up_non_fatal_config_message\n"));
580
581     if (config_error.duplicate_segment) {
582         for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
583             if (config_error.duplicate_segment & (1 << slotno)) {
584                 lm_queue_message(WARNING_MSG, "duplicate segment in lane: to slot: to",
585                                 'A' + slotno / MAX_SLOT_COUNT,
586                                 slotno % MAX_SLOT_COUNT);
587             }
588         }
589     }
590
591     if (config_error.missing_segment) {
592         for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
593             if (config_error.missing_segment & (1 << slotno)) {
594                 lm_queue_message(WARNING_MSG, "missing segment for Device Adapter in lane: to slot: to",
595                                 'A' + slotno / MAX_SLOT_COUNT,
596                                 slotno % MAX_SLOT_COUNT);
597             }
598         }
599     }
600 }

```



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/lmserver.c

DATE

5/23/89

PAGE #

TIME

6:14:45 pm

6/92

LINE #	SOURCE TEXT
601	}
602	}
603	}
604	if (config_error.no_pel_for_dab) {
605	for (slotno = 0; slotno < MAX_LANE_COUNT; ++slotno) {
606	if (config_error.no_pel_for_dab & (1 <= slotno)) {
607	lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d is not supported by a Pds Electronics Module",
608	'A' + slotno / MAX_SLOT_COUNT,
609	slotno % MAX_SLOT_COUNT);
610	}
611	}
612	}
613	if (config_error.no_pam_for_dab) {
614	for (slotno = 0; slotno < MAX_LANE_COUNT; ++slotno) {
615	if (config_error.no_pam_for_dab & (1 <= slotno)) {
616	lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d is not backed by Fast Pattern Memory",
617	'A' + slotno / MAX_SLOT_COUNT,
618	slotno % MAX_SLOT_COUNT);
619	}
620	}
621	}
622	}
623	}
624	}
625	if (config_error.illegal_duplicate_device) {
626	for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
627	if (config_error.illegal_duplicate_device & (1 <= slotno)) {
628	lm_queue_message(WARNING_MSG, "Non-identical duplicate Device Adapter in lane: %c slot: %d",
629	'A' + slotno / MAX_SLOT_COUNT,
630	slotno % MAX_SLOT_COUNT);
631	}
632	}
633	}
634	}
635	if (config_error.device_too_large) {
636	for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
637	if (config_error.device_too_large & (1 <= slotno)) {
638	lm_queue_message(WARNING_MSG, "Device Adapter in lane: %c slot: %d makes up an illegal device size (> %d units)",
639	'A' + slotno / MAX_SLOT_COUNT,
640	slotno % MAX_SLOT_COUNT, MAX_UNIT_COUNT);
641	}
642	}
643	}
644	}
645	if (config_error.illegal_pel_stacking) {
646	for (slotno = 0; slotno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++slotno) {
647	if (config_error.illegal_pel_stacking & (1 <= slotno)) {
648	lm_queue_message(WARNING_MSG, "Illegal Pds Electronics Module stacking for Device Adapter in lane: %c slot: %d",
649	'A' + slotno / MAX_SLOT_COUNT,
650	slotno % MAX_SLOT_COUNT, MAX_UNIT_COUNT);
651	}
652	}
653	}
654	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/network.c

DATE 5/23/89  
TIME 6:14:46 pm

PAGE #  
1/93

LINE #	SOURCE TEXT
1	/* SCCS ID: network.c rev 3.1.1, 4/24/89 at 07:53:22 */
2	
3	#include "common.h"
4	#include "message.h"
5	#include "lnetwork.h"
6	#include "network.h"
7	
8	#ifdef DEBUG
9	#define DPRINTF(x) (void)printf x
10	else
11	#define DPRINTF(x) /* do nothing */
12	#endif
13	
14	end_queue_message()
15	{
16	u_short errors;
17	u_short warnings;
18	u_short type;
19	char    str[MAX_MESSAGE];
20	char    *ptr;
21	char    c;
22	
23	/* verify that we have not put anything in the output buffer except
24	the answer.
25	*/
26	if (LM_BYTES_IN_BUFFER(lm_global_conn_ptr) != 4) {
27	DPRINTF(("ASSERT: error count is not the first one in out buffer\n"));
28	}
29	
30	lm_message_types(&errors, &warnings);
31	if (errors+warnings) {
32	DPRINTF(("there are %d errors and warnings\n", errors+warnings));
33	}
34	
35	lm_put_int(errors + warnings);
36	
37	while (lm_dequeue_message(&type, str) == SUCCESS) {
38	lm_put_char(type);
39	
40	ptr = str;
41	while (c = *ptr++) {
42	lm_put_char(c);
43	}
44	
45	DPRINTF(("\"%s\""),
46	lm_put_char("\0"),
47	}
48	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/nextuser.c

DATE 5/23/89  
TIME 6:14:46 pm

PAGE #  
1/94

```

1  /* SCSS_ID: nextuser.c rev 3.1, 4/24/89 at 07:53:25 */
2
3  #include "common.h"
4  #include "fifo.h"
5  #include "network.h"
6  #include "cpu.h"
7
8  #ifdef DEBUG
9  #define DPRINTF(x) (void)printf x
10 #else
11 #define DPRINTF(x) /* do nothing */
12 #endif
13
14 extern CONNECTION *table_of_conns[ ];
15 extern u_long lm_dead_user_mask;
16
17 u_short
18 lm_give_me_the_next_user(user)
19 u_char *user;
20 {
21     #ifdef MODELER
22     struct fifo_entry fifo;
23     #endif
24     u_char next_dead_user;
25
26     if (lm_dead_user_mask != 0) {
27         CPU_DISABLE_INTERRUPTS;
28         for (next_dead_user = 0; next_dead_user < MAX_USERS; next_dead_user++) {
29             if (lm_dead_user_mask & (1 << next_dead_user)) {
30                 abort_user_punbar( next_dead_user );
31                 close_connection_for_server( table_of_conns[ next_dead_user ] );
32             }
33         }
34         lm_dead_user_mask = 0;
35         CPU_ENABLE_INTERRUPTS;
36     }
37     #ifdef MODELER
38     u_short ret;
39
40     while (1) {
41         ret = lm_choose_connection(user);
42         if (ret == PENDING) {
43             if (lm_send_reply(table_of_conns[user]) != SUCCESS) {
44                 DPRINTF(("error in lm_send_reply()\n"));
45                 return(FAILURE);
46             }
47         }
48         else {
49             return(ret);
50         }
51     }
52 #else
53     fifo.fifo_no = RX_FIFO;
54     if (fifo_get(fifo) == SUCCESS) {
55         *user = fifo.user;
56         return(SUCCESS);
57     }
58     else {
59         return(FAILURE);
60     }
61 #endif
62 #endif
63 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/pac\_util.c

DATE 5/23/89  
TIME 6:14:46 pm

PAGE #  
1/95

```

1  /* SCCS ID: pac_util.c rev 1.1, 4/24/89 at 07:53.29 */
2
3  #include "common.h"
4  #include "vrtx.h"
5  #include "tmg.h"
6  #include "pac.h"
7  #include "pac_def.h"
8
9  extern TMG *tmgptr;
10
11 #ifdef DEBUG
12 #define DPRINTF(x) (void)printf x
13 #else
14 #define DPRINTF(x) /* do nothing */
15 #endif
16
17 /*
18  * int backplane_reset()
19  *
20  * INPUT: none
21  * OUTPUT: returns code = SUCCESS or FAILURE
22  * DESCRIPTION: Resets TMG and all four lanes.
23  */
24 int
25 backplane_reset()
26 {
27     long i;
28
29     /* Initialize Timing Generator */
30     DPRINTF(("inside backplane_reset\n"));
31     *REG(0) = 0x00; /* drive TMG reset and all lane resets */
32     *REG(1) = 0x00;
33     *REG(2) = 0x2c; /* 25 MHz clock */
34     *REG(3) = 0x0ff; /* Select divide by 2 */
35     *REG(4) = 0x000; /* Select fast ramps, all lanes */
36     *REG(5) = 0x62; /* Test mode 1, 11ns dlydly */
37     *REG(6) = 0x07; /* 10ns dump, 1st clock */
38     *REG(7) = 0x00; /* sample width = 1 clock cycle */
39     *REG(16) = 0x07; /* minimum threshold EDGE[0] */
40     *REG(17) = 0x08; /* EDGE[1] */
41     *REG(18) = 0x09; /* EDGE[2] */
42     *REG(19) = 0x0a; /* EDGE[3] */
43     *REG(20) = 0x0b; /* EDGE[4] */
44     *REG(21) = 0x0c; /* EDGE[5] */
45     *REG(22) = 0x07; /* SAMPLE trigger */
46     *REG(23) = 0x07; /* SAMPLE */
47
48     if(pac_clock_off() != SUCCESS)
49         return(FAILURE);
50     if(pac_clock_on() != SUCCESS)
51         return(FAILURE);
52
53     tmgptr->tmg_reset1 = 1; /* remove TMG reset */
54
55     if(tmg_play(16) != SUCCESS) {
56         DPRINTF(("Power-on play failed.\n"));
57         return(FAILURE);
58     }
59
60     *REG(5) = 0x22; /* No test mode 2, 11ns dlydly */
61
62     if(pac_clock_off() != SUCCESS)
63         return(FAILURE);
64     if(pac_clock_on() != SUCCESS)
65         return(FAILURE);
66
67     DPRINTF(("wait 1\n"));
68     for (i = 0; i < 10000; ++i)
69         ;
70
71     if(pac_clock_off() != SUCCESS)
72         return(FAILURE);
73
74     DPRINTF(("wait 1\n"));
75     for (i = 0; i < 10000; ++i)
76         ;
77
78     *REG(0) = 0x30; /* remove all resets */
79
80     if (tmgptr->lane_intr != 0) {
81         DPRINTF(("clearing lane_intr\n"));
82         (void)lm_write_probe(0x8c100280, 0);
83         (void)lm_write_probe(0x9c100280, 0);
84         (void)lm_write_probe(0xac100280, 0);
85         (void)lm_write_probe(0xdc100280, 0);
86     }
87
88     if (tmgptr->lane_intr != 0) {
89         DPRINTF(("lane_intr stuck\n"));
90         return(FAILURE);
91     }
92
93     DPRINTF(("backplane reset OK\n"));
94     return(SUCCESS);
95 }
96
97 /*
98  * int pac_clock_on()
99  *
100  * INPUT: none
101  * OUTPUT: returns SUCCESS or FAILURE
102  * DESCRIPTION: Turns on pattern clock.
103  */
104 int
105 pac_clock_on()
106 {
107     u_long start;
108
109     DPRINTF(("inside pac_clock_on\n"));
110     tmgptr->clock_sync_clear1 = 1;
111     tmgptr->clock_enable = 1;
112     start = lm_time();
113     while(!tmgptr->clock_on)
114     {
115         if((lm_time() - start) > 50) /* 5ms timeout */
116         {
117             DPRINTF(("Unable to turn on clock.\n"));
118             return(FAILURE);
119         }
120     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/pac\_util.c

DATE 5/23/89  
TIME 6:14:46 pm

PAGE #  
2/96

LINE # SOURCE TEXT

```
121 }  
122 return(SUCCESS);  
123 }  
124  
125  
126 int pac_clock_off()  
127 *  
128 * INPUT: none  
129 * OUTPUT: returns SUCCESS or FAILURE  
130 * DESCRIPTION: Turns off pattern clock.  
131 */  
132 int  
133 pac_clock_off()  
134 {  
135     u_long start;  
136  
137     tmpttr->clock_enable = 0;  
138     start = lm_time();  
139     while(tmpttr->clock_on)  
140     {  
141         if((lm_time() - start) > 50) /* 5ms timeout */  
142         {  
143             DPRINTF(("Unable to turn off clock.\n"));  
144             return(FAILURE);  
145         }  
146     }  
147     tmpttr->clock_sync_clear1 = 0;  
148     return(SUCCESS);  
149 }  
150
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/private.c	DATE 5/23/89	PAGE # 1/97
LINE #	SOURCE TEXT			
1	/* SCCS ID: private.c rev 3.1, 4/24/89 at 07:53:32 */			
2				
3	#include "device.h"			
4	#include "message.h"			
5	#include "hardware.h"			
6	#include "eeprom.h"			
7	#include "laser.h"			
8				
9	#define MAX_LINE_LENGTH 256			
10				
11	evaluate_private(def_ptr, instance, ident_change,			
12	ident_inconsistent_pins, changed_dac)			
13	DEVICE_SPEC *def_ptr;			
14	INSTANCE_INFO *instance;			
15	PTRN_BITS_LONGWORD *ident_change;			
16	PTRN_BITS_LONGWORD *ident_inconsistent_pins;			
17	u_char *changed_dac;			
18	{			
19	EXTRA_DEVICE_SPEC *extra_def_ptr;			
20	PIN_INFO *pin_info;			
21	PIN_SPEC *pin_def;			
22	UWB_OFFSET *uwb_ptr;			
23	u_long inst_block_number[MAX_LANE_COUNT];			
24	u_long seg_wd_addr[MAX_LANE_COUNT];			
25	u_short eval_index = 0;			
26	i;			
27	short pin_number;			
28	u_char pin_value;			
29	u_char old_pin_value;			
30	u_char junk1;			
31	u_char junk2;			
32	u_char unitno;			
33	u_char wordno;			
34	u_char bitno;			
35	u_char seg_driving_to_s;			
36	{			
37	DPRINTF(("inside evaluate_private\n"));			
38	extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;			
39				
40	if (set_edge_and_sample_setting(extra_def_ptr,			
41	(u_long)RESOLUTION_05_NS,			
42	(u_long)MAX_SAMPLE_RANGE) == FAILURE) {			
43	return(FAILURE);			
44	}			
45				
46	update_ptrn_loaded(instance, def_ptr);			
47				
48	/* Process the data pins and modify the measurement pattern accordingly */			
49	pin_number = instance->first_data_pin_index;			
50	instance->first_data_pin_index = -1;			
51	while (pin_number != -1) {			
52	{			
53	uwb_ptr = seg_to_short_offset(pin_number);			
54	unitno = uwb_ptr->unitno;			
55	wordno = uwb_ptr->wordno;			
56	bitno = uwb_ptr->bitno;			
57	pin_info = <instance->pin_info_table[pin_number];			
58	pin_info->input_pin_is_linked = FALSE;			
59	pin_value = pin_info->old_filtered;			
60	set_pin_value(<instance->pin_value,			
61	unitno, wordno, bitno, pin_value);			
62	set_measurement_pattern(instance, <def_ptr->pin_table[pin_number],			
63	unitno, wordno, bitno, pin_value,			
64	&junk1, &junk2);			
65	pin_number = pin_info->next_input_pin_index;			
66	}			
67				
68	/* Process the eval pins and modify the measurement pattern accordingly */			
69	pin_number = instance->first_eval_pin_index;			
70	instance->first_eval_pin_index = -1;			
71	while (pin_number != -1) {			
72	{			
73	uwb_ptr = seg_to_short_offset(pin_number);			
74	unitno = uwb_ptr->unitno;			
75	wordno = uwb_ptr->wordno;			
76	bitno = uwb_ptr->bitno;			
77	pin_info = <instance->pin_info_table[pin_number];			
78	pin_info->input_pin_is_linked = FALSE;			
79	pin_value = pin_info->old_filtered;			
80	set_pin_value(<instance->pin_value,			
81	unitno, wordno, bitno, pin_value);			
82	/* Save the EVAL changes to find the output delays */			
83	if (eval_index < MAX_EVAL_CHANNELS) {			
84	gbl_eval_pin_number[eval_index] = pin_number;			
85	gbl_eval_pin_value[eval_index] = pin_value;			
86	++eval_index;			
87	}			
88	else {			
89	in_queue_message(MEMORY_MSG, "internal error: not enough room to store eval changes; delay number might be incorrect");			
90	}			
91	set_measurement_pattern(instance, <def_ptr->pin_table[pin_number],			
92	unitno, wordno, bitno, pin_value,			
93	&junk1, &junk2);			
94	pin_number = pin_info->next_input_pin_index;			
95	}			
96				
97	/* If there are any eval changes, then run a measurement cycle */			
98	if (eval_index != 0) {			
99	{			
100	DPRINTF(("run measurement cycle for eval changes\n"));			
101				
102	/* Write ENDWB to next to last address */			
103	write_pattern(instance,			
104	<instance->unit_addr[instance->cur_unit_addr_index + 1 & 1][0],			
105	instance->hndwb_loaded);			
106				
107	/* Write Consistent set */			
108	write_pattern(instance,			
109				
110				
111				
112				
113				
114				
115				
116				
117				
118				
119				
120				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/private.c

DATE 5/23/89  
TIME 6:14:47 pm

PAGE #  
2/98

```

LINE #          SOURCE TEXT
121             *instance->unit_addr(instance->cur_unit_addr_index)[0],
122             instance->ptrn_loaded);
123
124         set_seq_end_bit(instance,
125             instance->lane_addr,
126             FALSE,
127             seq_end_addr,
128             inst_block_number);
129
130         if (play_ptrn_seq(instance, (u_long)PTRN_PLAY_TIMEOUT,
131             changed_gac) == FAILURE) {
132             return(FAILURE);
133         }
134
135         if (get_result(def_ptr, instance, ident_change,
136             EVAL_EVENT, gbl_eval_pin_number,
137             eval_index, &any_driving_to_z) == FAILURE) {
138             return(FAILURE);
139         }
140
141         remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
142
143         /* Get the delays from eval pins to outputs and put them in PIN_INFO */
144         for (i = 0; i < eval_index; ++i) {
145             get_delay(def_ptr, instance, ident_change,
146                 ident_inconsistent_pins,
147                 gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
148         }
149
150         clear_ptrn_bits((char *)ident_change,
151             dab_list(instance->dab_info_index)->unit_count);
152
153
154         /* Process the STORE pin changes */
155         for (pin_number = instance->first_store_pin_index,
156             instance->first_store_pin_index = -1,
157             pin_number != -1;
158             pin_number = pin_info->next_input_pin_index) {
159             update_ptrn_loaded(instance, def_ptr);
160
161             gbl_eval_pin_number[0] = pin_number;
162
163             unitno = uwb_ptr->unitno;
164             wordno = uwb_ptr->wordno;
165             bitno = uwb_ptr->bitno;
166
167             pin_info = instance->pin_info_table[pin_number];
168             pin_def = def_ptr->pin_table[pin_number];
169
170             pin_info->input_pin_is_linked = FALSE;
171             pin_value = pin_info->old_filtered;
172
173             if (pin_def->direction == IN) {
174                 /* INPUT STORE change */
175
176                 old_pin_value = read_pin_value(instance->pin_pin_value,
177                     unitno, wordno, bitno);
178
179                 set_pin_value(instance->pin_pin_value,
180                     unitno, wordno, bitno, pin_value);
181
182                 switch (pin_def->clk_format) {
183                     case NRZ:
184                         if (input_pin_transition(old_pin_value, pin_value,
185                             pin_info->uninitialized_pin) == NO_TRANSITION) {
186                             /* This is an error because the host should have filtered the
187                                * transition.
188                                * On second thought it is NOT an error because the host only
189                                * filters transitions to exactly the same value.
190                                */
191                             DPRINTF("No transition on IN STORE NRZ pin %s\n", pin_def->pin_name);
192                             continue;
193                         }
194
195                         if (pin_value & (LOGIC_0 | LOGIC_50 | LOGIC_Z0))
196                             reset_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno);
197                         else
198                             set_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno);
199                         break;
200
201                     case R1:
202                         if (input_pin_transition(old_pin_value, pin_value,
203                             pin_info->uninitialized_pin) != RISE_TRANSITION) {
204                             DPRINTF("No transition on IN STORE R1 pin %s\n", pin_def->pin_name);
205                             continue;
206                         }
207
208                         /* Enable the R1 clock on the measurement pattern (ptrn_loaded) */
209                         reset_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno);
210                         break;
211
212                     case R0:
213                         if (input_pin_transition(old_pin_value, pin_value,
214                             pin_info->uninitialized_pin) != FALL_TRANSITION) {
215                             DPRINTF("No transition on IN STORE R0 pin %s\n", pin_def->pin_name);
216                             continue;
217                         }
218
219                         /* Enable the RZ clock on the measurement pattern (ptrn_loaded) */
220                         set_ptrn_bit(instance->ptrn_loaded(unitno), wordno, bitno);
221                         break;
222
223                     default:
224                         /* Internal error: illegal pin type in device.h */
225                         in_queue_message(ERROR_MSG, "internal error: illegal pin type in device.h");
226                         continue;
227                 }
228             }
229
230             else {
231                 /* IO STORE change */
232                 /* Note: IO store pin can only have NRZ format */
233
234                 old_pin_value = read_pin_value(instance->last_sample_value,
235                     unitno, wordno, bitno);
236             }
237
238
239
240

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/private.c

DATE 5/23/89  
TIME 6:14:47 pm

PAGE #  
3/99

```

LINE # SOURCE TEXT
241 set_pin_value(instance->pin_value,
242 unitno, wordno, bitno, pin_value);
243
244 if (!io_pin_transition(old_pin_value, pin_value,
245 pin_info->uninitialized_pin) == PTN_TRANSITION) {
246
247     DPRINTF("No transition on IO STORE pin %s\n", pin_def->pin_name);
248     continue;
249 }
250
251 set_measurement_pattern(instance, pin_def,
252 unitno, wordno, bitno, pin_value,
253 ajunk1, ajunk2);
254
255 }
256
257 write_pattern(instance,
258 instance->unit_addr(instance->cur_unit_addr_index + 1 + 1)[0],
259 instance->inst_block_loaded);
260
261 write_pattern(instance,
262 instance->unit_addr(instance->cur_unit_addr_index)[0],
263 instance->ptrn_loaded);
264
265 /* Disable the EL/EL check for the next evaluation */
266 switch (pin_def->ela_format) {
267 case EL1:
268     set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
269     break;
270 case R0:
271     reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
272     break;
273 default:
274     break;
275 }
276
277 set_seq_end_bit(instance,
278 instance->lane_addr,
279 FALSE,
280 seq_end_addr,
281 inst_block_number);
282
283 if (play_ptrn_seq(instance, (u_long)PTN_PLAY_TIMEOUT,
284 changed_dac) == FAILURE) {
285     return(FAILURE);
286 }
287
288 if (get_result(def_ptr, instance, ident_change,
289 STORE_EVENT, gbl_eval_pin_number,
290 1, any_driving_to_1) == FAILURE) {
291     return(FAILURE);
292 }
293
294 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
295
296 get_delay(def_ptr, instance, ident_change, ident_inconsistent_pins,
297 (u_short)pin_number, pin_value);
298
299 clear_ptrn_bits((char *)ident_change,
300 dab_list(instance->dab_info_index->unit_count);
301
302 pin_info->uninitialized_pin = FALSE;
303
304 return(SUCCESS);
305
306 }
307
308 purge_ptrn(instance)
309 INSTANCE_INFO *instance;
310 {
311     DAB_INFO *dab_ptr;
312     short ptrn_count;
313     u_char dummy_ptrn_count;
314     u_char lane0;
315     u_char i;
316
317     DPRINTF("inside purge_ptrn\n");
318
319     dab_ptr = dab_list(instance->dab_info_index);
320
321     return_all_ptrn_block(instance);
322
323     setup_gbl_dummy_ptrn(dab_ptr);
324
325     for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
326         instance->fb_block_size[lane0] = 0;
327
328         instance->lane_addr[lane0].max_addr = 0;
329         instance->lane_addr[lane0].prev_max_addr = 0;
330         instance->lane_addr[lane0].inst_unit_addr = 0;
331         instance->lane_addr[lane0].seq_block_addr = 0;
332     }
333
334     instance->has_history = FALSE;
335     instance->pattern_count = 0;
336     instance->common_pattern_count = 0;
337     instance->static_pattern_count = 0;
338     instance->enab = 1;
339     instance->enab = 1;
340
341     if (allocate_initial_block(instance) == FAILURE)
342         return(FAILURE);
343
344     /* set the sequence start address on each lane */
345     for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0)
346         instance->seq_start_addr[lane0] =
347             instance->lane_addr[lane0].max_addr - BLOCK_ADDR_INC;
348
349     /* For PRIVATE devices we only have 2 patterns: ENDSER_LOADED and
350      * PTN_LOADED.
351      */
352     ptrn_count = 2 * dab_ptr->unit_count_per_lane;
353
354     dummy_ptrn_count = 0;
355     if (ptrn_count <= SEQ_END_LATENCY) {
356         dummy_ptrn_count = (SEQ_END_LATENCY - ptrn_count +
357             dab_ptr->unit_count_per_lane) /
358             dab_ptr->unit_count_per_lane;
359     }
360 }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/private.c

\*

DATE 5/23/89  
TIME 6:14:47 pm

PAGE #  
4/100

```

LINE # SOURCE TEXT
361 for (i = 0; i < dummy_ptr_count; ++i) {
362     write_pattern(instance,
363         instance->unit_addr(instance->cur_unit_addr_index)[0],
364         &dummy_ptr);
365
366     if (grow_pattern(instance) == FAILURE)
367         return(FAILURE);
368 }
369
370 /* allocate 2 patterns for MEMBERS */
371 if (grow_pattern(instance) == FAILURE)
372     return(FAILURE);
373
374 return(SUCCESS);
375 }
376
377 update_ptr_loaded(instance, def_ptr)
378 INSTANCE_INFO *instance;
379 DEVICE_SPEC *def_ptr;
380 {
381     EXTRA_DEVICE_SPEC *extra_def_ptr;
382     DAB_INFO *dab_ptr;
383     PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
384     PTRN_BITS_LONGWORD *sampled_data_ptr;
385     PTRN_BITS_LONGWORD *sampled_hiz_ptr;
386     PTRN_BITS_LONGWORD *sampled_unk_ptr;
387     PTRN_BITS_LONGWORD *sim_data_ptr;
388     PTRN_BITS_LONGWORD *sim_hiz_ptr;
389     PTRN_BITS_LONGWORD *sim_unk_ptr;
390     PTRN_BITS_LONGWORD *ident_outputs_ptr;
391     PTRN_BITS_LONGWORD *ident_ios_ptr;
392     u_char total_unit;
393     u_char unit;
394     u_char word;
395
396     /* Modify the ptrn_loaded to include the value of the last sample:
397      * for OUTPUT pins --> take the last_sample_value
398      * for IO pins --> take the combination of sim_pin_value and
399      * last_sample_value.
400     */
401
402     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
403     dab_ptr = dab_list(instance->dab_info_index);
404     total_unit = dab_ptr->unit_count;
405
406     ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
407     sampled_data_ptr = (PTRN_BITS_LONGWORD *)
408         instance->last_sample_value.data;
409     sampled_hiz_ptr = (PTRN_BITS_LONGWORD *)
410         instance->last_sample_value.hiz;
411     sampled_unk_ptr = (PTRN_BITS_LONGWORD *)
412         instance->last_sample_value.unknown;
413     sim_data_ptr = (PTRN_BITS_LONGWORD *)
414         instance->sim_pin_value.data;
415     sim_hiz_ptr = (PTRN_BITS_LONGWORD *)
416         instance->sim_pin_value.hiz;
417     sim_unk_ptr = (PTRN_BITS_LONGWORD *)
418         instance->sim_pin_value.unknown;
419     ident_outputs_ptr = extra_def_ptr->ident_outputs;
420     ident_ios_ptr = extra_def_ptr->ident_ios;
421
422     for (unit = 0; unit < total_unit; ++unit) {
423         for (word = 0; word < 3; ++word) {
424
425             /* OUTPUT pins.
426              * Drive the output pins to 0 always.
427              */
428             ptrn_loaded_ptr->word[word] =
429                 (ptrn_loaded_ptr->word[word] & "ident_outputs_ptr->word[word]);
430
431             /* IO pins.
432              * Drive the IO pins to the combination of sim_pin_value and
433              * last_sample_value as follows:
434
435              SIN      ANY      Z      U      ANY
436
437              SAMPLE   ANY      Z      U
438
439              RESULT   $1      $2      $5      $3      $4
440
441              $1 --> use sample data
442              $2 --> use sample data
443              $3 --> use sim data
444              $4 --> use the opposite value of sample data
445              $5 --> use sample data
446
447              ptrn_loaded_ptr->word[word] =
448                  (ptrn_loaded_ptr->word[word] & "ident_ios_ptr->word[word]) |
449                  (ident_ios_ptr->word[word] &
450                      {
451                          /* Result $2 */
452                          ((sampled_hiz_ptr->word[word] & sim_hiz_ptr->word[word]) &
453                              sampled_data_ptr->word[word]) |
454                          /* Result $5 */
455                          ((sampled_hiz_ptr->word[word] & sim_unk_ptr->word[word]) &
456                              sampled_data_ptr->word[word]) |
457                          /* Result $3 */
458                          ((sampled_hiz_ptr->word[word] &
459                              (sim_hiz_ptr->word[word] | sim_unk_ptr->word[word])) &
460                              sim_data_ptr->word[word]) |
461                          /* Result $4 */
462                          (sampled_unk_ptr->word[word] & "sampled_data_ptr->word[word]) |
463                          /* Result $1 */
464                          ((sampled_hiz_ptr->word[word] | sampled_unk_ptr->word[word]) &
465                              sampled_data_ptr->word[word])
466                      });
467
468             ++ptrn_loaded_ptr;
469             ++sampled_data_ptr;
470
471         }
472     }
473 }
474
475 }
476
477 }
478
479 }
480

```

1523

5,353,243

1524

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/private.c	DATE 5/23/89	PAGE # 5/101
TIME 6:14:47 pm				
LINE #	SOURCE TEXT			
481	**sampled_hiz_ptr;			
482	**sampled_unk_ptr;			
483	**sin_data_ptr;			
484	**sin_hiz_ptr;			
485	**sin_unk_ptr;			
486	**idest_outputs_ptr;			
487	**idest_los_ptr;			
488	)			
489	)			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/profile.c

DATE	5/23/89	PAGE #
TIME	6:14:47 pm	1/102

```

1  /* SCCS_ID: profile.c rev 1.1. 4/24/89 at 07:53:35 */
2
3  #include "common.h"
4  #include "device.h"
5  #include "message.h"
6  #include "hardware.h"
7  #include "lm_rd_wr.h"
8  #include "mod_err.h"
9  #include "avaram.h"
10 #include "eeprom.h"
11 #include "lmsvr.h"
12 #include "lnetwork.h"
13 #include "network.h"
14 #include "profile.h"
15
16 PROF_DATA prof_data[MAX_FUNCTION_COUNT];
17 u_short xindex;
18 u_char initialized_profile = FALSE;
19 extern long save_pc;
20
21 profile_clear()
22 {
23     u_short i;
24
25     for (i = 0; i < xindex; ++i) {
26         prof_data[i].count = 0;
27     }
28 }
29
30 profile_incr_count()
31 {
32     register u_long pc;
33     register short start;
34     register short end;
35     register short mid;
36
37     if (initialized_profile == FALSE)
38         return;
39
40     pc = save_pc;
41     start = 0;
42     end = xindex - 1;
43
44     mid = (start + end) / 2;
45     while (start <= end) {
46         if (pc < prof_data[mid].function_addr) {
47             end = mid - 1;
48         }
49         else {
50             start = mid + 1;
51         }
52         mid = (start + end) / 2;
53     }
54
55     while (1) {
56         if ((pc >= prof_data[end].function_addr) &&
57             (pc < prof_data[end+1].function_addr)) {
58             break;
59         }
60         end++;
61     }
62
63     ++prof_data[end].count;
64 }
65
66 profile_dump_count()
67 {
68     short i;
69     long total;
70     u_long total_count_mark;
71
72     total = 0;
73     total_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
74     lm_put_int(0);
75
76     for (i = 0; i < xindex; ++i) {
77         if (prof_data[i].count != 0) {
78             ++total;
79             lm_put_int(prof_data[i].function_addr);
80             lm_put_int(prof_data[i].count);
81         }
82     }
83
84     LM_PUT_LONG_AT_MARK(total_count_mark, lm_global_conn_ptr, total);
85 }

```

Copyright 1989  
Logic Modeling Systems

HEADER FILE  
lm1000/profile.h

DATE 5/23/89  
TIME 6:14:47 pm

PAGE #  
1/103

LINE #	HEADER TEXT
1	/* SCSS_ID: profile.h rev 3.1, 4/24/89 at 07:53:38 */
2	
3	#define MAX_FUNCTION_COUNT 2000
4	
5	typedef struct {
6	u_long func;       addr;
7	u_long count;
8	} PROF_DATA;
9	
10	extern PROF_DATA prof_data[MAX_FUNCTION_COUNT];
11	extern u_short xindex;
12	extern u_char initialized_profile;

Copyright 1989  
Logic Modeling SystemsSOURCE PROGRAM  
lm1000/profile2.cDATE 5/23/89 PAGE #  
TIME 6:14:47 pm 1/104

LINE # SOURCE TEXT

```
1  /* SCCS ID: profile2.c rev 1.1, 4/24/89 at 07:53:41 */
2
3  #include "common.h"
4  #include "profile.h"
5
6  profile_ t()
7  {
8      xindex = 0;
9      prof_data[xindex++].function_addr = 0;
10     prof_data[xindex++].function_addr = 0x00050000;
11     prof_data[xindex++].function_addr = 0x00050026;
12     prof_data[xindex++].function_addr = 0x00050030;
13     prof_data[xindex++].function_addr = 0x00050048;
14     prof_data[xindex++].function_addr = 0x00050076;
15     prof_data[xindex++].function_addr = 0x0005007A;
16     prof_data[xindex++].function_addr = 0x00050092;
17     prof_data[xindex++].function_addr = 0x000500AA;
18     prof_data[xindex++].function_addr = 0x000500DA;
19     prof_data[xindex++].function_addr = 0x000500DE;
20     prof_data[xindex++].function_addr = 0x0005013A;
21     prof_data[xindex++].function_addr = 0x0005014E;
22     prof_data[xindex++].function_addr = 0x0005017A;
23     prof_data[xindex++].function_addr = 0x0005018E;
24     prof_data[xindex++].function_addr = 0x0005019A;
25     prof_data[xindex++].function_addr = 0x000501A8;
26     prof_data[xindex++].function_addr = 0x000501B8;
27     prof_data[xindex++].function_addr = 0x000501E8;
28     prof_data[xindex++].function_addr = 0x00050204;
29     prof_data[xindex++].function_addr = 0x00050230;
30     prof_data[xindex++].function_addr = 0x00050238;
31     prof_data[xindex++].function_addr = 0x0005024C;
32     prof_data[xindex++].function_addr = 0x00050258;
33     prof_data[xindex++].function_addr = 0x00050268;
34     prof_data[xindex++].function_addr = 0x00050298;
35     prof_data[xindex++].function_addr = 0x000502B4;
36     prof_data[xindex++].function_addr = 0x000502E4;
37     prof_data[xindex++].function_addr = 0x000502FC;
38     prof_data[xindex++].function_addr = 0x00050314;
39     prof_data[xindex++].function_addr = 0x00050330;
40     prof_data[xindex++].function_addr = 0x00050336;
41     prof_data[xindex++].function_addr = 0x00050342;
42     prof_data[xindex++].function_addr = 0x0005038C;
43     prof_data[xindex++].function_addr = 0x00050398;
44     prof_data[xindex++].function_addr = 0x000503A2;
45     prof_data[xindex++].function_addr = 0x000503CC;
46     prof_data[xindex++].function_addr = 0x0005041E;
47     prof_data[xindex++].function_addr = 0x00050480;
48     prof_data[xindex++].function_addr = 0x00050494;
49     prof_data[xindex++].function_addr = 0x000504AA;
50     prof_data[xindex++].function_addr = 0x000504AB;
51     prof_data[xindex++].function_addr = 0x000504B2;
52     prof_data[xindex++].function_addr = 0x000504BA;
53     prof_data[xindex++].function_addr = 0x000504BC;
54     prof_data[xindex++].function_addr = 0x000504F8;
55     prof_data[xindex++].function_addr = 0x00050504;
56     prof_data[xindex++].function_addr = 0x0005073E;
57     prof_data[xindex++].function_addr = 0x000507FE;
58     prof_data[xindex++].function_addr = 0x00050986;
59     prof_data[xindex++].function_addr = 0x00050A02;
60     prof_data[xindex++].function_addr = 0x00050A92;
61     prof_data[xindex++].function_addr = 0x00050AD6;
62     prof_data[xindex++].function_addr = 0x00050B8C;
63     prof_data[xindex++].function_addr = 0x00050C58;
64     prof_data[xindex++].function_addr = 0x00050E56;
65     prof_data[xindex++].function_addr = 0x00050E80;
66     prof_data[xindex++].function_addr = 0x00050FA0;
67     prof_data[xindex++].function_addr = 0x000510DE;
68     prof_data[xindex++].function_addr = 0x000511A2;
69     prof_data[xindex++].function_addr = 0x00051266;
70     prof_data[xindex++].function_addr = 0x0005131E;
71     prof_data[xindex++].function_addr = 0x0005152A;
72     prof_data[xindex++].function_addr = 0x000515EE;
73     prof_data[xindex++].function_addr = 0x0005166E;
74     prof_data[xindex++].function_addr = 0x000516A4;
75     prof_data[xindex++].function_addr = 0x0005192C;
76     prof_data[xindex++].function_addr = 0x00051AA4;
77     prof_data[xindex++].function_addr = 0x00051AD4;
78     prof_data[xindex++].function_addr = 0x00051AE8;
79     prof_data[xindex++].function_addr = 0x00051F34;
80     prof_data[xindex++].function_addr = 0x00051F88;
81     prof_data[xindex++].function_addr = 0x0005201E;
82     prof_data[xindex++].function_addr = 0x00052088;
83     prof_data[xindex++].function_addr = 0x000520B0;
84     prof_data[xindex++].function_addr = 0x000520FE;
85     prof_data[xindex++].function_addr = 0x00052144;
86     prof_data[xindex++].function_addr = 0x00052170;
87     prof_data[xindex++].function_addr = 0x000521F4;
88     prof_data[xindex++].function_addr = 0x0005221C;
89     prof_data[xindex++].function_addr = 0x00052244;
90     prof_data[xindex++].function_addr = 0x000522A2;
91     prof_data[xindex++].function_addr = 0x000523E0;
92     prof_data[xindex++].function_addr = 0x00052536;
93     prof_data[xindex++].function_addr = 0x00052786;
94     prof_data[xindex++].function_addr = 0x0005283A;
95     prof_data[xindex++].function_addr = 0x00052A7E;
96     prof_data[xindex++].function_addr = 0x00052C88;
97     prof_data[xindex++].function_addr = 0x00052D4A;
98     prof_data[xindex++].function_addr = 0x00052F68;
99     prof_data[xindex++].function_addr = 0x00052F90;
100    prof_data[xindex++].function_addr = 0x00052FBA;
101    prof_data[xindex++].function_addr = 0x00052FE2;
102    prof_data[xindex++].function_addr = 0x0005301E;
103    prof_data[xindex++].function_addr = 0x0005339C;
104    prof_data[xindex++].function_addr = 0x00053682;
105    prof_data[xindex++].function_addr = 0x000536DA;
106    prof_data[xindex++].function_addr = 0x00053704;
107    prof_data[xindex++].function_addr = 0x0005372C;
108    prof_data[xindex++].function_addr = 0x0005375A;
109    prof_data[xindex++].function_addr = 0x00053C08;
110    prof_data[xindex++].function_addr = 0x00053DEE;
111    prof_data[xindex++].function_addr = 0x00053E0C;
112    prof_data[xindex++].function_addr = 0x00053E1C;
113    prof_data[xindex++].function_addr = 0x00053E38;
114    prof_data[xindex++].function_addr = 0x00053E58;
115    prof_data[xindex++].function_addr = 0x0005418A;
116    prof_data[xindex++].function_addr = 0x000542E4;
117    prof_data[xindex++].function_addr = 0x0005432A;
118    prof_data[xindex++].function_addr = 0x000544EA;
119    prof_data[xindex++].function_addr = 0x0005470E;
120    prof_data[xindex++].function_addr = 0x0005475E;
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/profile2.c	DATE 5/23/89	PAGE # 2/105
LINE #		SOURCE TEXT		
121	prof_data(xindex++)	function_addr = 0x000547A8		
122	prof_data(xindex++)	function_addr = 0x000547F0		
123	prof_data(xindex++)	function_addr = 0x00054848		
124	prof_data(xindex++)	function_addr = 0x00054908		
125	prof_data(xindex++)	function_addr = 0x000549F0		
126	prof_data(xindex++)	function_addr = 0x00054AA0		
127	prof_data(xindex++)	function_addr = 0x00054B6C		
128	prof_data(xindex++)	function_addr = 0x00054C32		
129	prof_data(xindex++)	function_addr = 0x0005530E		
130	prof_data(xindex++)	function_addr = 0x00055804		
131	prof_data(xindex++)	function_addr = 0x00055952		
132	prof_data(xindex++)	function_addr = 0x00055A24		
133	prof_data(xindex++)	function_addr = 0x00055C92		
134	prof_data(xindex++)	function_addr = 0x00055E80		
135	prof_data(xindex++)	function_addr = 0x00055F7E		
136	prof_data(xindex++)	function_addr = 0x000562AA		
137	prof_data(xindex++)	function_addr = 0x0005644E		
138	prof_data(xindex++)	function_addr = 0x000567CE		
139	prof_data(xindex++)	function_addr = 0x000568AA		
140	prof_data(xindex++)	function_addr = 0x00056958		
141	prof_data(xindex++)	function_addr = 0x00056A64		
142	prof_data(xindex++)	function_addr = 0x00056B3C		
143	prof_data(xindex++)	function_addr = 0x00056BA0		
144	prof_data(xindex++)	function_addr = 0x00056B9A		
145	prof_data(xindex++)	function_addr = 0x00056BE2		
146	prof_data(xindex++)	function_addr = 0x00056C46		
147	prof_data(xindex++)	function_addr = 0x00056C4E		
148	prof_data(xindex++)	function_addr = 0x00056CA4		
149	prof_data(xindex++)	function_addr = 0x00056D8A		
150	prof_data(xindex++)	function_addr = 0x00056E22		
151	prof_data(xindex++)	function_addr = 0x000572C2		
152	prof_data(xindex++)	function_addr = 0x0005731C		
153	prof_data(xindex++)	function_addr = 0x000576FC		
154	prof_data(xindex++)	function_addr = 0x00057950		
155	prof_data(xindex++)	function_addr = 0x0005795A		
156	prof_data(xindex++)	function_addr = 0x00057A16		
157	prof_data(xindex++)	function_addr = 0x00057ABE		
158	prof_data(xindex++)	function_addr = 0x00057B62		
159	prof_data(xindex++)	function_addr = 0x00057CBE		
160	prof_data(xindex++)	function_addr = 0x00057D3A		
161	prof_data(xindex++)	function_addr = 0x00057D3E		
162	prof_data(xindex++)	function_addr = 0x00057D5A		
163	prof_data(xindex++)	function_addr = 0x00057D72		
164	prof_data(xindex++)	function_addr = 0x00057DDE		
165	prof_data(xindex++)	function_addr = 0x00057F94		
166	prof_data(xindex++)	function_addr = 0x00058274		
167	prof_data(xindex++)	function_addr = 0x00058378		
168	prof_data(xindex++)	function_addr = 0x000585C4		
169	prof_data(xindex++)	function_addr = 0x000587FA		
170	prof_data(xindex++)	function_addr = 0x0005891E		
171	prof_data(xindex++)	function_addr = 0x00058AC2		
172	prof_data(xindex++)	function_addr = 0x00058B58		
173	prof_data(xindex++)	function_addr = 0x00058B80		
174	prof_data(xindex++)	function_addr = 0x00058C0A		
175	prof_data(xindex++)	function_addr = 0x00058C7A		
176	prof_data(xindex++)	function_addr = 0x00058E0A		
177	prof_data(xindex++)	function_addr = 0x00058E3A		
178	prof_data(xindex++)	function_addr = 0x00058E42		
179	prof_data(xindex++)	function_addr = 0x00058EDA		
180	prof_data(xindex++)	function_addr = 0x00058F70		
181	prof_data(xindex++)	function_addr = 0x00059160		
182	prof_data(xindex++)	function_addr = 0x000593BA		
183	prof_data(xindex++)	function_addr = 0x0005950C		
184	prof_data(xindex++)	function_addr = 0x00059590		
185	prof_data(xindex++)	function_addr = 0x000597E4		
186	prof_data(xindex++)	function_addr = 0x0005AACE		
187	prof_data(xindex++)	function_addr = 0x0005A35E		
188	prof_data(xindex++)	function_addr = 0x0005A5BC		
189	prof_data(xindex++)	function_addr = 0x0005A8BC		
190	prof_data(xindex++)	function_addr = 0x0005A95C		
191	prof_data(xindex++)	function_addr = 0x0005AC12		
192	prof_data(xindex++)	function_addr = 0x0005AE3A		
193	prof_data(xindex++)	function_addr = 0x0005B254		
194	prof_data(xindex++)	function_addr = 0x0005B5CC		
195	prof_data(xindex++)	function_addr = 0x0005B73C		
196	prof_data(xindex++)	function_addr = 0x0005B7FC		
197	prof_data(xindex++)	function_addr = 0x0005C1EA		
198	prof_data(xindex++)	function_addr = 0x0005C6F0		
199	prof_data(xindex++)	function_addr = 0x0005C928		
200	prof_data(xindex++)	function_addr = 0x0005CB0A		
201	prof_data(xindex++)	function_addr = 0x0005D494		
202	prof_data(xindex++)	function_addr = 0x0005D76A		
203	prof_data(xindex++)	function_addr = 0x0005DC3C		
204	prof_data(xindex++)	function_addr = 0x0005DE08		
205	prof_data(xindex++)	function_addr = 0x0005E13C		
206	prof_data(xindex++)	function_addr = 0x0005E678		
207	prof_data(xindex++)	function_addr = 0x0005E870		
208	prof_data(xindex++)	function_addr = 0x0005E9F8		
209	prof_data(xindex++)	function_addr = 0x0005EEA6		
210	prof_data(xindex++)	function_addr = 0x0005F454		
211	prof_data(xindex++)	function_addr = 0x0005F5E8		
212	prof_data(xindex++)	function_addr = 0x0005F8BC		
213	prof_data(xindex++)	function_addr = 0x0005FB90		
214	prof_data(xindex++)	function_addr = 0x0005FD9C		
215	prof_data(xindex++)	function_addr = 0x0005FF5A		
216	prof_data(xindex++)	function_addr = 0x00060160		
217	prof_data(xindex++)	function_addr = 0x00060344		
218	prof_data(xindex++)	function_addr = 0x00060398		
219	prof_data(xindex++)	function_addr = 0x00060878		
220	prof_data(xindex++)	function_addr = 0x00060906		
221	prof_data(xindex++)	function_addr = 0x000610F4		
222	prof_data(xindex++)	function_addr = 0x000612D6		
223	prof_data(xindex++)	function_addr = 0x000613C4		
224	prof_data(xindex++)	function_addr = 0x000614C0		
225	prof_data(xindex++)	function_addr = 0x0006151E		
226	prof_data(xindex++)	function_addr = 0x0006152E		
227	prof_data(xindex++)	function_addr = 0x0006153E		
228	prof_data(xindex++)	function_addr = 0x0006154C		
229	prof_data(xindex++)	function_addr = 0x0006155C		
230	prof_data(xindex++)	function_addr = 0x0006156C		
231	prof_data(xindex++)	function_addr = 0x0006157C		
232	prof_data(xindex++)	function_addr = 0x00061590		
233	prof_data(xindex++)	function_addr = 0x0006160A		
234	prof_data(xindex++)	function_addr = 0x00061638		
235	prof_data(xindex++)	function_addr = 0x0006168E		
236	prof_data(xindex++)	function_addr = 0x000616CA		
237	prof_data(xindex++)	function_addr = 0x00061774		
238	prof_data(xindex++)	function_addr = 0x00061C32		
239	prof_data(xindex++)	function_addr = 0x00061CCE		
240	prof_data(xindex++)	function_addr = 0x00061D76		

5,353,243

1533

1534

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/profile2.c

DATE

5/23/89

PAGE #

TIME

6:14:47 pm

3/106

LINE # SOURCE TEXT

```

241 prof_data[xindex++].function_addr = 0x00061DD0,
242 prof_data[xindex++].function_addr = 0x00061E56,
243 prof_data[xindex++].function_addr = 0x00061E7C,
244 prof_data[xindex++].function_addr = 0x00062106,
245 prof_data[xindex++].function_addr = 0x00062294,
246 prof_data[xindex++].function_addr = 0x00062342,
247 prof_data[xindex++].function_addr = 0x000623B6,
248 prof_data[xindex++].function_addr = 0x00062852,
249 prof_data[xindex++].function_addr = 0x000628F4,
250 prof_data[xindex++].function_addr = 0x00062E90,
251 prof_data[xindex++].function_addr = 0x000633FE,
252 prof_data[xindex++].function_addr = 0x00063520,
253 prof_data[xindex++].function_addr = 0x00063558,
254 prof_data[xindex++].function_addr = 0x00063632,
255 prof_data[xindex++].function_addr = 0x000639A0,
256 prof_data[xindex++].function_addr = 0x00063A30,
257 prof_data[xindex++].function_addr = 0x00063AD2,
258 prof_data[xindex++].function_addr = 0x00063BC8,
259 prof_data[xindex++].function_addr = 0x00063C8E,
260 prof_data[xindex++].function_addr = 0x00063CB6,
261 prof_data[xindex++].function_addr = 0x00063D08,
262 prof_data[xindex++].function_addr = 0x00063DB2,
263 prof_data[xindex++].function_addr = 0x000640A0,
264 prof_data[xindex++].function_addr = 0x000641A8,
265 prof_data[xindex++].function_addr = 0x0006439E,
266 prof_data[xindex++].function_addr = 0x000643F8,
267 prof_data[xindex++].function_addr = 0x0006446A,
268 prof_data[xindex++].function_addr = 0x0006470A,
269 prof_data[xindex++].function_addr = 0x00065ADA,
270 prof_data[xindex++].function_addr = 0x00065B0C,
271 prof_data[xindex++].function_addr = 0x00065BEC,
272 prof_data[xindex++].function_addr = 0x00066044,
273 prof_data[xindex++].function_addr = 0x000661D4,
274 prof_data[xindex++].function_addr = 0x00066B3A,
275 prof_data[xindex++].function_addr = 0x00066CBA,
276 prof_data[xindex++].function_addr = 0x0006727C,
277 prof_data[xindex++].function_addr = 0x00067444,
278 prof_data[xindex++].function_addr = 0x000674A2,
279 prof_data[xindex++].function_addr = 0x00067500,
280 prof_data[xindex++].function_addr = 0x00067E76,
281 prof_data[xindex++].function_addr = 0x00068040,
282 prof_data[xindex++].function_addr = 0x000681E8,
283 prof_data[xindex++].function_addr = 0x00068D1C,
284 prof_data[xindex++].function_addr = 0x000693FC,
285 prof_data[xindex++].function_addr = 0x0006961E,
286 prof_data[xindex++].function_addr = 0x00069D3A,
287 prof_data[xindex++].function_addr = 0x00069E74,
288 prof_data[xindex++].function_addr = 0x0006A000,
289 prof_data[xindex++].function_addr = 0x0006A442,
290 prof_data[xindex++].function_addr = 0x0006A78A,
291 prof_data[xindex++].function_addr = 0x0006A85E,
292 prof_data[xindex++].function_addr = 0x0006A8AA,
293 prof_data[xindex++].function_addr = 0x0006A9CD,
294 prof_data[xindex++].function_addr = 0x0006A9D6,
295 prof_data[xindex++].function_addr = 0x0006AB4C,
296 prof_data[xindex++].function_addr = 0x0006ABE0,
297 prof_data[xindex++].function_addr = 0x0006AC18,
298 prof_data[xindex++].function_addr = 0x0006D854,
299 prof_data[xindex++].function_addr = 0x0006D9DE,
300 prof_data[xindex++].function_addr = 0x0006DD50,
301 prof_data[xindex++].function_addr = 0x0006DF30,
302 prof_data[xindex++].function_addr = 0x0006DFE6,
303 prof_data[xindex++].function_addr = 0x0006E9FC,
304 prof_data[xindex++].function_addr = 0x0006ECFA,
305 prof_data[xindex++].function_addr = 0x0006ED38,
306 prof_data[xindex++].function_addr = 0x0006ED4C,
307 prof_data[xindex++].function_addr = 0x0006EFF8,
308 prof_data[xindex++].function_addr = 0x0006F2FC,
309 prof_data[xindex++].function_addr = 0x0006F35C,
310 prof_data[xindex++].function_addr = 0x00070190,
311 prof_data[xindex++].function_addr = 0x000702E8,
312 prof_data[xindex++].function_addr = 0x00070D9C,
313 prof_data[xindex++].function_addr = 0x00071122,
314 prof_data[xindex++].function_addr = 0x00071210,
315 prof_data[xindex++].function_addr = 0x000712C4,
316 prof_data[xindex++].function_addr = 0x00071690,
317 prof_data[xindex++].function_addr = 0x00071688,
318 prof_data[xindex++].function_addr = 0x00071994,
319 prof_data[xindex++].function_addr = 0x000719E8,
320 prof_data[xindex++].function_addr = 0x00071A08,
321 prof_data[xindex++].function_addr = 0x00071A22,
322 prof_data[xindex++].function_addr = 0x00071B14,
323 prof_data[xindex++].function_addr = 0x00071B4E,
324 prof_data[xindex++].function_addr = 0x00071BBA,
325 prof_data[xindex++].function_addr = 0x00071C16,
326 prof_data[xindex++].function_addr = 0x00071CEE,
327 prof_data[xindex++].function_addr = 0x00071D68,
328 prof_data[xindex++].function_addr = 0x00071D96,
329 prof_data[xindex++].function_addr = 0x00071DD2,
330 prof_data[xindex++].function_addr = 0x00071E44,
331 prof_data[xindex++].function_addr = 0x00071E4A,
332 prof_data[xindex++].function_addr = 0x00071E12,
333 prof_data[xindex++].function_addr = 0x00071F3E,
334 prof_data[xindex++].function_addr = 0x00071F88,
335 prof_data[xindex++].function_addr = 0x000720AA,
336 prof_data[xindex++].function_addr = 0x00072496,
337 prof_data[xindex++].function_addr = 0x00072524,
338 prof_data[xindex++].function_addr = 0x000725CA,
339 prof_data[xindex++].function_addr = 0x00072796,
340 prof_data[xindex++].function_addr = 0x000728FC,
341 prof_data[xindex++].function_addr = 0x0007285E,
342 prof_data[xindex++].function_addr = 0x00073C8C,
343 prof_data[xindex++].function_addr = 0x00073D00,
344 prof_data[xindex++].function_addr = 0x00073DEA,
345 prof_data[xindex++].function_addr = 0x00073E76,
346 prof_data[xindex++].function_addr = 0x00073E94,
347 prof_data[xindex++].function_addr = 0x00073E3E,
348 prof_data[xindex++].function_addr = 0x00073F24,
349 prof_data[xindex++].function_addr = 0x00073FA6,
350 prof_data[xindex++].function_addr = 0x00074014,
351 prof_data[xindex++].function_addr = 0x0007406A,
352 prof_data[xindex++].function_addr = 0x000740FC,
353 prof_data[xindex++].function_addr = 0x0007431C,
354 prof_data[xindex++].function_addr = 0x000743A4,
355 prof_data[xindex++].function_addr = 0x000743F0,
356 prof_data[xindex++].function_addr = 0x00074426,
357 prof_data[xindex++].function_addr = 0x0007444C,
358 prof_data[xindex++].function_addr = 0x00074492,
359 prof_data[xindex++].function_addr = 0x000744C8,
360 prof_data[xindex++].function_addr = 0x000744FE,

```

1536

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM lm1000/profile2.c	DATE 5/23/89	PAGE # 4/107
--	-------------------------------------	-----------------	-----------------

LINE #	SOURCE TEXT
361	prof_data(xindex++) .function_addr = 0x00074534;
362	prof_data(xindex++) .function_addr = 0x00074566;
363	prof_data(xindex++) .function_addr = 0x00074598;
364	prof_data(xindex++) .function_addr = 0x000745CA;
365	prof_data(xindex++) .function_addr = 0x000745FC;
366	prof_data(xindex++) .function_addr = 0x0007462E;
367	prof_data(xindex++) .function_addr = 0x00074660;
368	prof_data(xindex++) .function_addr = 0x000747D0;
369	prof_data(xindex++) .function_addr = 0x00074830;
370	prof_data(xindex++) .function_addr = 0x0007486E;
371	prof_data(xindex++) .function_addr = 0x000748B6;
372	prof_data(xindex++) .function_addr = 0x00074978;
373	prof_data(xindex++) .function_addr = 0x00074B14;
374	prof_data(xindex++) .function_addr = 0x00074BFA;
375	prof_data(xindex++) .function_addr = 0x00074C30;
376	prof_data(xindex++) .function_addr = 0x00074C96;
377	prof_data(xindex++) .function_addr = 0x00074CD2;
378	prof_data(xindex++) .function_addr = 0x00074D0A;
379	prof_data(xindex++) .function_addr = 0x00074D58;
380	prof_data(xindex++) .function_addr = 0x00074D90;
381	prof_data(xindex++) .function_addr = 0x00074DC8;
382	prof_data(xindex++) .function_addr = 0x00074E04;
383	prof_data(xindex++) .function_addr = 0x00074E3A;
384	prof_data(xindex++) .function_addr = 0x00074EF4;
385	prof_data(xindex++) .function_addr = 0x00074F14;
386	prof_data(xindex++) .function_addr = 0x000750A8;
387	prof_data(xindex++) .function_addr = 0x000753B2;
388	prof_data(xindex++) .function_addr = 0x000753C8;
389	prof_data(xindex++) .function_addr = 0x0007544C;
390	prof_data(xindex++) .function_addr = 0x000754CC;
391	prof_data(xindex++) .function_addr = 0x000754FA;
392	prof_data(xindex++) .function_addr = 0x00075562;
393	prof_data(xindex++) .function_addr = 0x000755CA;
394	prof_data(xindex++) .function_addr = 0x000755F4;
395	prof_data(xindex++) .function_addr = 0x0007567C;
396	prof_data(xindex++) .function_addr = 0x000756B2;
397	prof_data(xindex++) .function_addr = 0x00075700;
398	prof_data(xindex++) .function_addr = 0x00075746;
399	prof_data(xindex++) .function_addr = 0x00075792;
400	prof_data(xindex++) .function_addr = 0x00075844;
401	prof_data(xindex++) .function_addr = 0x00075890;
402	prof_data(xindex++) .function_addr = 0x00075C6C;
403	prof_data(xindex++) .function_addr = 0x00075CD4;
404	prof_data(xindex++) .function_addr = 0x00075D32;
405	prof_data(xindex++) .function_addr = 0x00075D80;
406	prof_data(xindex++) .function_addr = 0x00075E1E;
407	prof_data(xindex++) .function_addr = 0x000764E4;
408	prof_data(xindex++) .function_addr = 0x00076720;
409	prof_data(xindex++) .function_addr = 0x00076858;
410	prof_data(xindex++) .function_addr = 0x00076866;
411	prof_data(xindex++) .function_addr = 0x000768D4;
412	prof_data(xindex++) .function_addr = 0x00076B70;
413	prof_data(xindex++) .function_addr = 0x00076B8E;
414	prof_data(xindex++) .function_addr = 0x00076B32;
415	prof_data(xindex++) .function_addr = 0x00076B7E;
416	prof_data(xindex++) .function_addr = 0x00076B3F;
417	prof_data(xindex++) .function_addr = 0x00076B76;
418	prof_data(xindex++) .function_addr = 0x00076BCE;
419	prof_data(xindex++) .function_addr = 0x00076548;
420	prof_data(xindex++) .function_addr = 0x00076612;
421	prof_data(xindex++) .function_addr = 0x0007667E;
422	prof_data(xindex++) .function_addr = 0x00076828;
423	prof_data(xindex++) .function_addr = 0x0007687A;
424	prof_data(xindex++) .function_addr = 0x00076A16;
425	prof_data(xindex++) .function_addr = 0x00076A6C;
426	prof_data(xindex++) .function_addr = 0x00076A82;
427	prof_data(xindex++) .function_addr = 0x00076B3C;
428	prof_data(xindex++) .function_addr = 0x00076B8A;
429	prof_data(xindex++) .function_addr = 0x00076C5A;
430	prof_data(xindex++) .function_addr = 0x00076E02;
431	prof_data(xindex++) .function_addr = 0x0007690E;
432	prof_data(xindex++) .function_addr = 0x0007698A;
433	prof_data(xindex++) .function_addr = 0x00076A6C;
434	prof_data(xindex++) .function_addr = 0x000767D0;
435	prof_data(xindex++) .function_addr = 0x000767DE;
436	prof_data(xindex++) .function_addr = 0x000767F4;
437	prof_data(xindex++) .function_addr = 0x00076A34;
438	prof_data(xindex++) .function_addr = 0x00076A84;
439	prof_data(xindex++) .function_addr = 0x00076AEE;
440	prof_data(xindex++) .function_addr = 0x00076E8A;
441	prof_data(xindex++) .function_addr = 0x00076AFC;
442	prof_data(xindex++) .function_addr = 0x000765DC;
443	prof_data(xindex++) .function_addr = 0x0007663C;
444	prof_data(xindex++) .function_addr = 0x0007668C;
445	prof_data(xindex++) .function_addr = 0x000766C6;
446	prof_data(xindex++) .function_addr = 0x000766EE;
447	prof_data(xindex++) .function_addr = 0x000767E0;
448	prof_data(xindex++) .function_addr = 0x00076830;
449	prof_data(xindex++) .function_addr = 0x00076856;
450	prof_data(xindex++) .function_addr = 0x00076890;
451	prof_data(xindex++) .function_addr = 0x000768B6;
452	prof_data(xindex++) .function_addr = 0x000768F0;
453	prof_data(xindex++) .function_addr = 0x00076916;
454	prof_data(xindex++) .function_addr = 0x0007693C;
455	prof_data(xindex++) .function_addr = 0x00076962;
456	prof_data(xindex++) .function_addr = 0x00076990;
457	prof_data(xindex++) .function_addr = 0x000769C2;
458	prof_data(xindex++) .function_addr = 0x00076A08;
459	prof_data(xindex++) .function_addr = 0x00076A62;
460	prof_data(xindex++) .function_addr = 0x00076AFA;
461	prof_data(xindex++) .function_addr = 0x00076B5A;
462	prof_data(xindex++) .function_addr = 0x00076BA0;
463	prof_data(xindex++) .function_addr = 0x00076BDA;
464	prof_data(xindex++) .function_addr = 0



1538

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/profile2.c

DATE 5/23/89  
TIME 6:14:47 pm

PAGE #  
5/108

SOURCE TEXT

```
481 prof_data[xindex++].function_addr = 0x0007C462;
482 prof_data[xindex++].function_addr = 0x0007C46C;
483 prof_data[xindex++].function_addr = 0x0007C470;
484 prof_data[xindex++].function_addr = 0x0007C47E;
485 prof_data[xindex++].function_addr = 0x0007C568;
486 prof_data[xindex++].function_addr = 0x0007C64C;
487 prof_data[xindex++].function_addr = 0x0007C69C;
488 prof_data[xindex++].function_addr = 0x0007C6CA;
489 prof_data[xindex++].function_addr = 0x0007C6E8;
490 prof_data[xindex++].function_addr = 0x0007C702;
491 prof_data[xindex++].function_addr = 0x0007C74C;
492 prof_data[xindex++].function_addr = 0x0007C7C2;
493 prof_data[xindex++].function_addr = 0x0007C7E8;
494 prof_data[xindex++].function_addr = 0x0007CEE4;
495 prof_data[xindex++].function_addr = 0x0007D24C;
496 prof_data[xindex++].function_addr = 0x0007E22E;
497 prof_data[xindex++].function_addr = 0x0007E46E;
498 prof_data[xindex++].function_addr = 0x0007E5CA;
499 prof_data[xindex++].function_addr = 0x0007E5C6;
500 prof_data[xindex++].function_addr = 0x0007E7AC;
501 prof_data[xindex++].function_addr = 0x0007F77C;
502 prof_data[xindex++].function_addr = 0x0007F458;
503 prof_data[xindex++].function_addr = 0x0007F756;
504 prof_data[xindex++].function_addr = 0x0007FA54;
505 prof_data[xindex++].function_addr = 0x0007FD52;
506 prof_data[xindex++].function_addr = 0x0007FDD8;
507 prof_data[xindex++].function_addr = 0x0008038C;
508 prof_data[xindex++].function_addr = 0x000804AC;
509 prof_data[xindex++].function_addr = 0x000805FA;
510 prof_data[xindex++].function_addr = 0x000807D0;
511 prof_data[xindex++].function_addr = 0x00081A68;
512 prof_data[xindex++].function_addr = 0x00081BFE;
513 prof_data[xindex++].function_addr = 0x00081DEE;
514 prof_data[xindex++].function_addr = 0x00081E9C;
515 prof_data[xindex++].function_addr = 0x00081ED2;
516 prof_data[xindex++].function_addr = 0x00081F78;
517 prof_data[xindex++].function_addr = 0x000823E2;
518 prof_data[xindex++].function_addr = 0x00082800;
519 prof_data[xindex++].function_addr = 0x00082884;
520 prof_data[xindex++].function_addr = 0x0008282A;
521 prof_data[xindex++].function_addr = 0x00082A00;
522 prof_data[xindex++].function_addr = 0x00082AC6;
523 prof_data[xindex++].function_addr = 0x00082C1A;
524 prof_data[xindex++].function_addr = 0x00082C46;
525 prof_data[xindex++].function_addr = 0x00083252;
526 prof_data[xindex++].function_addr = 0x00083620;
527 prof_data[xindex++].function_addr = 0x000837CC;
528 prof_data[xindex++].function_addr = 0x00083C78;
529 prof_data[xindex++].function_addr = 0x00083E4A;
530 prof_data[xindex++].function_addr = 0x00083FF4;
531 prof_data[xindex++].function_addr = 0x00085B3C;
532 prof_data[xindex++].function_addr = 0x00085B78;
533 prof_data[xindex++].function_addr = 0x00085CCE;
534 prof_data[xindex++].function_addr = 0x00086048;
535 prof_data[xindex++].function_addr = 0x0008614A;
536 prof_data[xindex++].function_addr = 0x0008622A;
537 prof_data[xindex++].function_addr = 0x0008634E;
538 prof_data[xindex++].function_addr = 0x000863A0;
539 prof_data[xindex++].function_addr = 0x000863E8;
540 prof_data[xindex++].function_addr = 0x0008649A;
541 prof_data[xindex++].function_addr = 0x00086740;
542 prof_data[xindex++].function_addr = 0x0008697C;
543 prof_data[xindex++].function_addr = 0x00086C42;
544 prof_data[xindex++].function_addr = 0x00086C88;
545 prof_data[xindex++].function_addr = 0x00087104;
546 prof_data[xindex++].function_addr = 0x000871CC;
547 prof_data[xindex++].function_addr = 0x000872DC;
548 prof_data[xindex++].function_addr = 0x00087376;
549 prof_data[xindex++].function_addr = 0x00087466;
550 prof_data[xindex++].function_addr = 0x00087506;
551 prof_data[xindex++].function_addr = 0x0008758E;
552 prof_data[xindex++].function_addr = 0x00087644;
553 prof_data[xindex++].function_addr = 0x0008770A;
554 prof_data[xindex++].function_addr = 0x00087788;
555 prof_data[xindex++].function_addr = 0x00087D48;
556 prof_data[xindex++].function_addr = 0x00087E12;
557 prof_data[xindex++].function_addr = 0x00087EF4;
558 prof_data[xindex++].function_addr = 0x00087F04;
559 prof_data[xindex++].function_addr = 0x00087F2C;
560 prof_data[xindex++].function_addr = 0x00087F64;
561 prof_data[xindex++].function_addr = 0x00087F74;
562 prof_data[xindex++].function_addr = 0x00087F84;
563 prof_data[xindex++].function_addr = 0x00087F94;
564 prof_data[xindex++].function_addr = 0x00087FA4;
565 prof_data[xindex++].function_addr = 0x00087FB4;
566 prof_data[xindex++].function_addr = 0x00087FC4;
567 prof_data[xindex++].function_addr = 0x00087FD4;
568 prof_data[xindex++].function_addr = 0x00087FE4;
569 prof_data[xindex++].function_addr = 0x00087FF4;
570 prof_data[xindex++].function_addr = 0x00088004;
571 prof_data[xindex++].function_addr = 0x00088014;
572 prof_data[xindex++].function_addr = 0x00088024;
573 prof_data[xindex++].function_addr = 0x00088034;
574 prof_data[xindex++].function_addr = 0x00088044;
575 prof_data[xindex++].function_addr = 0x00088054;
576 prof_data[xindex++].function_addr = 0x00088064;
577 prof_data[xindex++].function_addr = 0x00088074;
578 prof_data[xindex++].function_addr = 0x00088082;
579 prof_data[xindex++].function_addr = 0x00088092;
580 prof_data[xindex++].function_addr = 0x000880AA;
581 prof_data[xindex++].function_addr = 0x000880BA;
582 prof_data[xindex++].function_addr = 0x000880CA;
583 prof_data[xindex++].function_addr = 0x000880DA;
584 prof_data[xindex++].function_addr = 0x000880E4;
585 prof_data[xindex++].function_addr = 0x000880F0;
586 prof_data[xindex++].function_addr = 0x000880F4;
587 prof_data[xindex++].function_addr = 0x000880F8;
588 prof_data[xindex++].function_addr = 0x000880FE;
589 prof_data[xindex++].function_addr = 0x000880FC;
590 prof_data[xindex++].function_addr = 0x000880FA;
591 prof_data[xindex++].function_addr = 0x000880F8;
592 prof_data[xindex++].function_addr = 0x000880F6;
593 prof_data[xindex++].function_addr = 0x000880F4;
594 prof_data[xindex++].function_addr = 0x000880F2;
595 prof_data[xindex++].function_addr = 0x000880F0;
596 prof_data[xindex++].function_addr = 0x000880EE;
597 prof_data[xindex++].function_addr = 0x000880E8;
598 prof_data[xindex++].function_addr = 0x000880E2;
599 prof_data[xindex++].function_addr = 0x000880D8;
600 prof_data[xindex++].function_addr = 0x000880D0;
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/profile2.c	DATE 5/23/89	PAGE # 6/109
TIME 6:14:47 pm				
LINE #	SOURCE TEXT			
601	prof_data(xindex++) .function_addr = 0x0008927E;			
602	prof_data(xindex++) .function_addr = 0x0008931C;			
603	prof_data(xindex++) .function_addr = 0x000893B8;			
604	prof_data(xindex++) .function_addr = 0x000895E4;			
605	prof_data(xindex++) .function_addr = 0x000895EC;			
606	prof_data(xindex++) .function_addr = 0x00089600;			
607	prof_data(xindex++) .function_addr = 0x00089612;			
608	prof_data(xindex++) .function_addr = 0x00089C6E;			
609	prof_data(xindex++) .function_addr = 0x0008A19C;			
610	prof_data(xindex++) .function_addr = 0x0008A312;			
611	prof_data(xindex++) .function_addr = 0x0008A354;			
612	prof_data(xindex++) .function_addr = 0x0008A376;			
613	prof_data(xindex++) .function_addr = 0x0008AD32;			
614	prof_data(xindex++) .function_addr = 0x0008AE7C;			
615	prof_data(xindex++) .function_addr = 0x0008B292;			
616	prof_data(xindex++) .function_addr = 0x0008B384;			
617	prof_data(xindex++) .function_addr = 0x0008B37C;			
618	prof_data(xindex++) .function_addr = 0x0008B456;			
619	prof_data(xindex++) .function_addr = 0x0008B77C;			
620	prof_data(xindex++) .function_addr = 0x0008B87A;			
621	prof_data(xindex++) .function_addr = 0x0008B916;			
622	prof_data(xindex++) .function_addr = 0x0008BA3A;			
623	prof_data(xindex++) .function_addr = 0x0008BAC2;			
624	prof_data(xindex++) .function_addr = 0x0008BC6A;			
625	prof_data(xindex++) .function_addr = 0x0008BD92;			
626	prof_data(xindex++) .function_addr = 0x0008BE3E;			
627	prof_data(xindex++) .function_addr = 0x0008BF72;			
628	prof_data(xindex++) .function_addr = 0x0008C1B4;			
629	prof_data(xindex++) .function_addr = 0x0008C29E;			
630	prof_data(xindex++) .function_addr = 0x0008C310;			
631	prof_data(xindex++) .function_addr = 0x0008C4EE;			
632	prof_data(xindex++) .function_addr = 0x0008C6D0;			
633	prof_data(xindex++) .function_addr = 0x0008CF86;			
634	prof_data(xindex++) .function_addr = 0x0008D19C;			
635	prof_data(xindex++) .function_addr = 0x0008D7AA;			
636	prof_data(xindex++) .function_addr = 0x0008DE76;			
637	prof_data(xindex++) .function_addr = 0x0008E1D6;			
638	prof_data(xindex++) .function_addr = 0x0008E746;			
639	prof_data(xindex++) .function_addr = 0x0008E78A;			
640	prof_data(xindex++) .function_addr = 0x0008E7A2;			
641	prof_data(xindex++) .function_addr = 0x0008E7E6;			
642	prof_data(xindex++) .function_addr = 0x0008E816;			
643	prof_data(xindex++) .function_addr = 0x0008EC38;			
644	prof_data(xindex++) .function_addr = 0x0008EDC2;			
645	prof_data(xindex++) .function_addr = 0x0008EE14;			
646	prof_data(xindex++) .function_addr = 0x0008EE40;			
647	prof_data(xindex++) .function_addr = 0x0008EEAA;			
648	prof_data(xindex++) .function_addr = 0x0008EF20;			
649	prof_data(xindex++) .function_addr = 0x0008EFAC;			
650	prof_data(xindex++) .function_addr = 0x0008F08C;			
651	prof_data(xindex++) .function_addr = 0x0008F150;			
652	prof_data(xindex++) .function_addr = 0x0008F27A;			
653	prof_data(xindex++) .function_addr = 0x0008F376;			
654	prof_data(xindex++) .function_addr = 0x0008F396;			
655	prof_data(xindex++) .function_addr = 0x0008F3BE;			
656	prof_data(xindex++) .function_addr = 0x0008F3A8;			
657	prof_data(xindex++) .function_addr = 0x0008F4D6;			
658	prof_data(xindex++) .function_addr = 0x0008F7E0;			
659	prof_data(xindex++) .function_addr = 0x0008FAE2;			
660	prof_data(xindex++) .function_addr = 0x0008FAB8;			
661	prof_data(xindex++) .function_addr = 0x0008FB3A;			
662	prof_data(xindex++) .function_addr = 0x0008FBC4;			
663	prof_data(xindex++) .function_addr = 0x00090308;			
664	prof_data(xindex++) .function_addr = 0x00090324;			
665	prof_data(xindex++) .function_addr = 0x00090340;			
666	prof_data(xindex++) .function_addr = 0x000903E0;			
667	prof_data(xindex++) .function_addr = 0x000904C4;			
668	prof_data(xindex++) .function_addr = 0x0009054E;			
669	prof_data(xindex++) .function_addr = 0x00090652;			
670	prof_data(xindex++) .function_addr = 0x00090770;			
671	prof_data(xindex++) .function_addr = 0x0009085E;			
672	prof_data(xindex++) .function_addr = 0x000909F2;			
673	prof_data(xindex++) .function_addr = 0x00090A6E;			
674	prof_data(xindex++) .function_addr = 0x00090ABE;			
675	prof_data(xindex++) .function_addr = 0x00090A7E;			
676	prof_data(xindex++) .function_addr = 0x00090868;			
677	prof_data(xindex++) .function_addr = 0x00090C10;			
678	prof_data(xindex++) .function_addr = 0x00090E76;			
679	prof_data(xindex++) .function_addr = 0x00091008;			
680	prof_data(xindex++) .function_addr = 0x00091380;			
681	prof_data(xindex++) .function_addr = 0x000913DE;			
682	prof_data(xindex++) .function_addr = 0x000914C8;			
683	prof_data(xindex++) .function_addr = 0x00091566;			
684	prof_data(xindex++) .function_addr = 0x00091510;			
685	prof_data(xindex++) .function_addr = 0x0009151A;			
686	prof_data(xindex++) .function_addr = 0x00091540;			
687	prof_data(xindex++) .function_addr = 0x00091698;			
688	prof_data(xindex++) .function_addr = 0x00091688;			
689	prof_data(xindex++) .function_addr = 0x00091760;			
690	prof_data(xindex++) .function_addr = 0x000917DC;			
691	prof_data(xindex++) .function_addr = 0x00091ADE;			
692	prof_data(xindex++) .function_addr = 0x00091B24;			
693	prof_data(xindex++) .function_addr = 0x00091B4A;			
694	prof_data(xindex++) .function_addr = 0x00091B5A;			
695	prof_data(xindex++) .function_addr = 0x00091B48;			
696	prof_data(xindex++) .function_addr = 0x00091B8C;			
697	prof_data(xindex++) .function_addr = 0x00091C1C;			
698	prof_data(xindex++) .function_addr = 0x00091C3C;			
699	prof_data(xindex++) .function_addr = 0x00091C6C;			
700	prof_data(xindex++) .function_addr = 0x00091E28;			
701	prof_data(xindex++) .function_addr = 0x00091F38;			
702	prof_data(xindex++) .function_addr = 0x00092086;			
703	prof_data(xindex++) .function_addr = 0x000921C0;			
704	prof_data(xindex++) .function_addr = 0x000922DC;			
705	prof_data(xindex++) .function_addr = 0x0009231C;			
706	prof_data(xindex++) .function_addr = 0x00092348;			
707	prof_data(xindex++) .function_addr = 0x0009237E;			
708	prof_data(xindex++) .function_addr = 0x0009239E;			
709	prof_data(xindex++) .function_addr = 0x000923F0;			
710	prof_data(xindex++) .function_addr = 0x00092434;			
711	prof_data(xindex++) .function_addr = 0x000924E2;			
712	prof_data(xindex++) .function_addr = 0x00092558;			
713	prof_data(xindex++) .function_addr = 0x00092582;			
714	prof_data(xindex++) .function_addr = 0x000925F8;			
715	prof_data(xindex++) .function_addr = 0x0009269C;			
716	prof_data(xindex++) .function_addr = 0x000926D2;			
717	prof_data(xindex++) .function_addr = 0x0009274E;			
718	prof_data(xindex++) .function_addr = 0x00092804;			
719	prof_data(xindex++) .function_addr = 0x00092862;			
720	prof_data(xindex++) .function_addr = 0x000928DE;			

5,353,243

1541

1542

Copyright 1989  
Logic Modeling SystemsSOURCE PROGRAM  
lm1000/profile2.cDATE 5/23/89  
TIME 6:14:47 pmPAGE #  
7/110

LINE # SOURCE TEXT

```

721 prof_data[xindex++].function_addr = 0x0009295A,
722 prof_data[xindex++].function_addr = 0x00092B1A,
723 prof_data[xindex++].function_addr = 0x00092B7A,
724 prof_data[xindex++].function_addr = 0x00092BD4,
725 prof_data[xindex++].function_addr = 0x00092C2A,
726 prof_data[xindex++].function_addr = 0x00092F66,
727 prof_data[xindex++].function_addr = 0x00093122,
728 prof_data[xindex++].function_addr = 0x000931C4,
729 prof_data[xindex++].function_addr = 0x00093250,
730 prof_data[xindex++].function_addr = 0x00093286,
731 prof_data[xindex++].function_addr = 0x000934D8,
732 prof_data[xindex++].function_addr = 0x0009356C,
733 prof_data[xindex++].function_addr = 0x00093570,
734 prof_data[xindex++].function_addr = 0x00093B60,
735 prof_data[xindex++].function_addr = 0x00094020,
736 prof_data[xindex++].function_addr = 0x00094062,
737 prof_data[xindex++].function_addr = 0x00094114,
738 prof_data[xindex++].function_addr = 0x0009414C,
739 prof_data[xindex++].function_addr = 0x00094AF4,
740 prof_data[xindex++].function_addr = 0x00094B10,
741 prof_data[xindex++].function_addr = 0x00094B2C,
742 prof_data[xindex++].function_addr = 0x00094B60,
743 prof_data[xindex++].function_addr = 0x00094BA0,
744 prof_data[xindex++].function_addr = 0x00094D88,
745 prof_data[xindex++].function_addr = 0x00094E7C,
746 prof_data[xindex++].function_addr = 0x00094F3E,
747 prof_data[xindex++].function_addr = 0x00094FBA,
748 prof_data[xindex++].function_addr = 0x00095020,
749 prof_data[xindex++].function_addr = 0x00095056,
750 prof_data[xindex++].function_addr = 0x0009507A,
751 prof_data[xindex++].function_addr = 0x000950BA,
752 prof_data[xindex++].function_addr = 0x000950DE,
753 prof_data[xindex++].function_addr = 0x00095210,
754 prof_data[xindex++].function_addr = 0x00095260,
755 prof_data[xindex++].function_addr = 0x000954CA,
756 prof_data[xindex++].function_addr = 0x000954E2,
757 prof_data[xindex++].function_addr = 0x000954FA,
758 prof_data[xindex++].function_addr = 0x0009551C,
759 prof_data[xindex++].function_addr = 0x0009553A,
760 prof_data[xindex++].function_addr = 0x00095554,
761 prof_data[xindex++].function_addr = 0x0009558E,
762 prof_data[xindex++].function_addr = 0x0009564E,
763 prof_data[xindex++].function_addr = 0x000956E2,
764 prof_data[xindex++].function_addr = 0x00095740,
765 prof_data[xindex++].function_addr = 0x000957BA,
766 prof_data[xindex++].function_addr = 0x00095824,
767 prof_data[xindex++].function_addr = 0x00095866,
768 prof_data[xindex++].function_addr = 0x000958E8,
769 prof_data[xindex++].function_addr = 0x00095980,
770 prof_data[xindex++].function_addr = 0x00095A18,
771 prof_data[xindex++].function_addr = 0x00095B34,
772 prof_data[xindex++].function_addr = 0x00095B9E,
773 prof_data[xindex++].function_addr = 0x00095C46,
774 prof_data[xindex++].function_addr = 0x00095CDE,
775 prof_data[xindex++].function_addr = 0x00095CF4,
776 prof_data[xindex++].function_addr = 0x00095D5C,
777 prof_data[xindex++].function_addr = 0x00095DF2,
778 prof_data[xindex++].function_addr = 0x00095E2A,
779 prof_data[xindex++].function_addr = 0x00095E6C,
780 prof_data[xindex++].function_addr = 0x00095E88,
781 prof_data[xindex++].function_addr = 0x00095EB8,
782 prof_data[xindex++].function_addr = 0x00095ED8,
783 prof_data[xindex++].function_addr = 0x00095F10,
784 prof_data[xindex++].function_addr = 0x00095F34,
785 prof_data[xindex++].function_addr = 0x00095F84,
786 prof_data[xindex++].function_addr = 0x00095FCC,
787 prof_data[xindex++].function_addr = 0x00096094,
788 prof_data[xindex++].function_addr = 0x000960C8,
789 prof_data[xindex++].function_addr = 0x00096114,
790 prof_data[xindex++].function_addr = 0x00096144,
791 prof_data[xindex++].function_addr = 0x00096174,
792 prof_data[xindex++].function_addr = 0x000961CC,
793 prof_data[xindex++].function_addr = 0x00096374,
794 prof_data[xindex++].function_addr = 0x0009639C,
795 prof_data[xindex++].function_addr = 0x0009651C,
796 prof_data[xindex++].function_addr = 0x000965D0,
797 prof_data[xindex++].function_addr = 0x000965F8,
798 prof_data[xindex++].function_addr = 0x000965B0,
799 prof_data[xindex++].function_addr = 0x000965D4,
800 prof_data[xindex++].function_addr = 0x00096808,
801 prof_data[xindex++].function_addr = 0x00096C0C,
802 prof_data[xindex++].function_addr = 0x00096CB4,
803 prof_data[xindex++].function_addr = 0x00096D9E,
804 prof_data[xindex++].function_addr = 0x00096E74,
805 prof_data[xindex++].function_addr = 0x00097080,
806 prof_data[xindex++].function_addr = 0x000970E8,
807 prof_data[xindex++].function_addr = 0x0009713C,
808 prof_data[xindex++].function_addr = 0x0009726C,
809 prof_data[xindex++].function_addr = 0x0009748C,
810 prof_data[xindex++].function_addr = 0x000974F8,
811 prof_data[xindex++].function_addr = 0x00097504,
812 prof_data[xindex++].function_addr = 0x00097640,
813 prof_data[xindex++].function_addr = 0x000976AE,
814 prof_data[xindex++].function_addr = 0x0009765C,
815 prof_data[xindex++].function_addr = 0x00097678,
816 prof_data[xindex++].function_addr = 0x000976A0,
817 prof_data[xindex++].function_addr = 0x000976AE,
818 prof_data[xindex++].function_addr = 0x000976CC,
819 prof_data[xindex++].function_addr = 0x000976E2,
820 prof_data[xindex++].function_addr = 0x000976FA,
821 prof_data[xindex++].function_addr = 0x00097712,
822 prof_data[xindex++].function_addr = 0x0009772C,
823 prof_data[xindex++].function_addr = 0x00097748,
824 prof_data[xindex++].function_addr = 0x00097768,
825 prof_data[xindex++].function_addr = 0x0009778A,
826 prof_data[xindex++].function_addr = 0x000977B8,
827 prof_data[xindex++].function_addr = 0x000977E4,
828 prof_data[xindex++].function_addr = 0x0009780C,
829 prof_data[xindex++].function_addr = 0x00097832,
830 prof_data[xindex++].function_addr = 0x00097850,
831 prof_data[xindex++].function_addr = 0x00097876,
832 prof_data[xindex++].function_addr = 0x0009789C,
833 prof_data[xindex++].function_addr = 0x000978CC,
834 prof_data[xindex++].function_addr = 0x000978F0,
835 prof_data[xindex++].function_addr = 0x00097914,
836 prof_data[xindex++].function_addr = 0x00097938,
837 prof_data[xindex++].function_addr = 0x0009795E,
838 prof_data[xindex++].function_addr = 0x0009798E,
839 prof_data[xindex++].function_addr = 0x0009799E,
840 prof_data[xindex++].function_addr = 0x000979B0,

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/profile2.c	DATE 5/23/89	PAGE # 8/111
LINE #	SOURCE TEXT			
841	prof_data[xindex++].function_addr = 0x000979DE;			
842	prof_data[xindex++].function_addr = 0x000979E0;			
843	prof_data[xindex++].function_addr = 0x000979E4;			
844	prof_data[xindex++].function_addr = 0x000979E8;			
845	prof_data[xindex++].function_addr = 0x000979EC;			
846	prof_data[xindex++].function_addr = 0x000979F0;			
847	prof_data[xindex++].function_addr = 0x000979F4;			
848	prof_data[xindex++].function_addr = 0x000979F8;			
849	prof_data[xindex++].function_addr = 0x000979FC;			
850	prof_data[xindex++].function_addr = 0x00097A00;			
851	prof_data[xindex++].function_addr = 0x00097A04;			
852	prof_data[xindex++].function_addr = 0x00097A08;			
853	prof_data[xindex++].function_addr = 0x00097A0C;			
854	prof_data[xindex++].function_addr = 0x00097A10;			
855	prof_data[xindex++].function_addr = 0x00097A14;			
856	prof_data[xindex++].function_addr = 0x00097A18;			
857	prof_data[xindex++].function_addr = 0x00097A1C;			
858	prof_data[xindex++].function_addr = 0x00097A20;			
859	prof_data[xindex++].function_addr = 0x00097A24;			
860	prof_data[xindex++].function_addr = 0x00097A28;			
861	prof_data[xindex++].function_addr = 0x00097A2C;			
862	prof_data[xindex++].function_addr = 0x00097A30;			
863	prof_data[xindex++].function_addr = 0x00097A34;			
864	prof_data[xindex++].function_addr = 0x00097A38;			
865	prof_data[xindex++].function_addr = 0x00097A3C;			
866	prof_data[xindex++].function_addr = 0x00097A40;			
867	prof_data[xindex++].function_addr = 0x00097A44;			
868	prof_data[xindex++].function_addr = 0x00097A48;			
869	prof_data[xindex++].function_addr = 0x00097A4C;			
870	prof_data[xindex++].function_addr = 0x00097A50;			
871	prof_data[xindex++].function_addr = 0x00097A54;			
872	prof_data[xindex++].function_addr = 0x00097A58;			
873	prof_data[xindex++].function_addr = 0x00097A5C;			
874	prof_data[xindex++].function_addr = 0x00097A60;			
875	prof_data[xindex++].function_addr = 0x00097A64;			
876	prof_data[xindex++].function_addr = 0x00097A68;			
877	prof_data[xindex++].function_addr = 0x00097A6C;			
878	prof_data[xindex++].function_addr = 0x00097A70;			
879	prof_data[xindex++].function_addr = 0x00097A74;			
880	prof_data[xindex++].function_addr = 0x00097A78;			
881	prof_data[xindex++].function_addr = 0x00097A7C;			
882	prof_data[xindex++].function_addr = 0x00097A80;			
883	prof_data[xindex++].function_addr = 0x00097A84;			
884	prof_data[xindex++].function_addr = 0x00097A88;			
885	prof_data[xindex++].function_addr = 0x00097A8C;			
886	prof_data[xindex++].function_addr = 0x00097A90;			
887	prof_data[xindex++].function_addr = 0x00097A94;			
888	prof_data[xindex++].function_addr = 0x00097A98;			
889	prof_data[xindex++].function_addr = 0x00097AA0;			
890	prof_data[xindex++].function_addr = 0x00097AA4;			
891	prof_data[xindex++].function_addr = 0x00097AA8;			
892	prof_data[xindex++].function_addr = 0x00097AAE;			
893	initialized_profile = TRUE;			
894				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
1/112

```

LINE # SOURCE TEXT
1  /* SCCS ID: ptrnhist.c rev 3.1, 4/24/89 at 07:53:47 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "a.h"
7  #include "lserver.h"
8  #include "protass.h"
9  #include "lnetwork.h"
10 #include "network.h"
11
12 #define MAX_LINE_LENGTH 128
13
14 #ifdef DEBUG
15 extern u_char loop_till_key;
16 extern u_long debug_key;
17 extern u_long debug_char;
18 #endif
19
20 evaluate_1_bit_per_pin(def_ptr, instance,
21                        ident_inconsistent_pins,
22                        temp_steady_state_result,
23                        ident_change,
24                        replay_count,
25                        changed_dec)
26
27 DEVICE_SPEC *def_ptr;
28 INSTANCE_INFO *instance;
29 PTRN_BITS LONGWORD *ident_inconsistent_pins;
30 FULL_VALUE *temp_steady_state_result;
31 PTRN_BITS LONGWORD *ident_change;
32 u_char replay_count;
33 u_char changed_dec;
34
35 EXTRA_DEVICE_SPEC *extra_def_ptr;
36 PIN_INFO *pin_info;
37 PIN_SPEC *pin_def;
38 PTRN_BITS *consistent_set(MAX_UNIT_COUNT);
39 UWB_OFFSET *uwb_ptr;
40 u_long last_block_number(MAX_LANE_COUNT);
41 u_long seq_wnd_addr(MAX_LANE_COUNT);
42 timer;
43 u_short eval_index = 0;
44 u_short i;
45 u_short pin_number;
46 u_char pin_value;
47 u_char old_pin_value;
48 u_char lcyddb_modified = FALSE;
49 u_char lcyddb_modified = FALSE;
50 u_char unitno;
51 u_char wordno;
52 u_char bitno;
53 u_char total_unit;
54 u_char any_driving_to_1;
55
56 DPRINTF(("inside evaluate_1_bit_per_pin\n"));
57
58 /* The timeout value is set with the assumption that we are running at
59 * 15MHz ("bus period").
60 * timeout = pattern play time + 1 second for feedback
61 *          = pattern_count * 8 microsec + 125000 * 8 microsec
62 *          = (pattern_count + 125000) * 8 microsec
63 *          = (pattern_count + 125000) * 8 / 1000 millisecc
64 */
65 timeout = (instance->pattern_count + PTRN_COUNT_FUDGE_FACTOR) >> 7;
66
67 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
68
69 if (set_edge_and_sample_setting(extra_def_ptr,
70                                (u_long)RESOLUTION_05_NS,
71                                (u_long)MAX_SAMPLE_RANGE) == FAILURE)
72     return(FAILURE);
73
74 total_unit = dab_list(instance->dab_info_index)->unit_count;
75
76 if (instance->is_fault == TRUE) {
77     /* If it is a FAULT then always write the LCYCHDB and LCYCHDB */
78     write_pattern(instance,
79                  &instance->lcychdb_addr[0],
80                  instance->lcychdb_loaded);
81
82     write_pattern(instance,
83                  &instance->lcychdb_addr[0],
84                  instance->lcychdb_loaded);
85 }
86 else {
87     /* If it is an INSTANCE then write the LCYCHDB and LCYCHDB if this
88     * instance has any faults.
89     */
90     if (instance->fault_count != 0) {
91         write_pattern(instance,
92                      &instance->lcychdb_addr[0],
93                      instance->lcychdb_loaded);
94
95         write_pattern(instance,
96                      &instance->lcychdb_addr[0],
97                      instance->lcychdb_loaded);
98     }
99 }
100
101 /* Process the data pins and modify the measurement pattern accordingly */
102 pin_number = instance->first_data_pin_index;
103 instance->first_data_pin_index = -1;
104 while (pin_number != -1) {
105     uwb_ptr = apn_to_short_offset(pin_number);
106     unitno = uwb_ptr->unitno;
107     wordno = uwb_ptr->wordno;
108     bitno = uwb_ptr->bitno;
109
110     pin_info = &instance->pin_info_table[pin_number];
111     pin_info->input_pin_is_linked = FALSE;
112     pin_value = pin_info->old_filtered;
113
114     set_pin_value(&instance->pin_value,
115                  unitno, wordno, bitno, pin_value);
116
117     set_measurement_patterns(instance, &def_ptr->pin_table[pin_number],
118                              unitno, wordno, bitno, pin_value,
119                              &lcychdb_modified, &lcychdb_modified);
120 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
2/113

```

121  LINE #          SOURCE TEXT
122      pin_number = pin_info->next_input_pin_index;
123  }
124  /* Process the eval pins and modify the measurement pattern accordingly */
125  pin_number = instance->first_eval_pin_index;
126  instance->first_eval_pin_index = -1;
127  while (pin_number != -1) {
128      /* Get the offset for this pin */
129      word_ptr = (u_int16_t *) &offset(pin_number);
130      unitno = word_ptr->unitno;
131      wordno = word_ptr->wordno;
132      bitno = word_ptr->bitno;
133      pin_info = instance->pin_info_table[pin_number];
134      pin_info->input_pin_is_linked = FALSE;
135      pin_value = pin_info->old_filtered;
136      set_pin_value(instance->pin_value,
137                  unitno, wordno, bitno, pin_value);
138      /* Save the EVAL changes to find the output delays */
139      if (eval_index < MAX_EVAL_CHANGES) {
140          gbl_eval_pin_number[eval_index] = pin_number;
141          gbl_eval_pin_value[eval_index] = pin_value;
142          ++eval_index;
143      }
144      else {
145          /* Internal error: not enough room to store eval changes; delay number might be incorrect */
146          lm_queue_message(MESSING_MSG, "internal error: not enough room to store eval changes; delay number might be incorrect");
147      }
148      set_measurement_patterns(instance, &def_ptr->pin_table[pin_number],
149                          unitno, wordno, bitno, pin_value,
150                          &lcychdb_modified, &lcycmdb_modified);
151      pin_number = pin_info->next_input_pin_index;
152  }
153  /* If there are any eval changes, then run a measurement cycle */
154  if (eval_index != 0) {
155      DPRINTF(("run measurement cycle for eval changes\n"));
156      if (lcychdb_modified == TRUE) {
157          write_patterns(instance,
158                      instance->lcychdb_addr[0],
159                      instance->lcychdb_loaded);
160          lcychdb_modified = FALSE;
161      }
162      if (lcycmdb_modified == TRUE) {
163          write_patterns(instance,
164                      instance->lcycmdb_addr[0],
165                      instance->lcycmdb_loaded);
166          lcycmdb_modified = FALSE;
167      }
168      write_patterns(instance,
169                  instance->unit_addr[instance->cur_unit_addr_index][0],
170                  instance->ptrs_loaded);
171      set_seq_end_bit(instance,
172                  instance->last_addr,
173                  FALSE,
174                  seq_end_addr,
175                  inst_block_number);
176      if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
177          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
178          return(FAILURE);
179      }
180      if (get_result(def_ptr, instance, idest_change,
181                  EVAL_EVENT, gbl_eval_pin_number,
182                  eval_index, &any_driving_to_1) == FAILURE) {
183          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
184          return(FAILURE);
185      }
186      for (i = 0; i < replay_count; ++i) {
187          if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
188              remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
189              return(FAILURE);
190          }
191          read_magic_full_sample_reg(instance, temp_steady_state_result);
192          check_consistency(instance,
193                      instance->last_sample_value,
194                      temp_steady_state_result,
195                      idest_inconsistent_pins,
196                      idest_change,
197                      total_unit);
198      }
199      #ifdef DEBUG
200      if (loop_till_key == TRUE) {
201          printf("looping in eval instance. EVAL changes\n");
202          debug_key = 0;
203          while (debug_key == 0) {
204              if (play_ptrn_seq(instance,
205                          timeout, &changed_dac) == FAILURE) {
206                  remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
207                  return(FAILURE);
208              }
209              if (debug_char == (u_long)'t') {
210                  loop_till_key = FALSE;
211                  printf("stop loop\n");
212              }
213              debug_key = 0;
214          }
215      }
216      #endif
217      if (any_driving_to_1 == TRUE) {
218          /* Use 2 bits per pin -> hard drive those IO STORE pins which
219          change from driving to 1.
220          */
221          if (def_ptr->device_type == PUBLIC) {
222              /* Write ENDEND_LOADED to previous pattern address.
223              */
224          }
225      }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89 PAGE #  
TIME 6:14:48 pm 3/114

```

LINE # SOURCE TEXT
241  * For PRIVATE devices we don't have to do anything since
242  * RMODE will always be written when we do the next evaluation.
243  */
244  write_ptrns(instance,
245      (instance->unit_addr[
246          instance->cur_unit_addr_index + 1 & 1][0],
247      instance->hndemb_loaded);
248
249  write_ptrns(instance,
250      (instance->unit_addr[instance->cur_unit_addr_index][0],
251      instance->last_consistent_set);
252
253  if (grow_ptrns(instance) == FAILURE)
254      return(FAILURE);
255  }
256
257  remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
258
259  /* Get the delays from eval pins to outputs and put them in PIN_INFO */
260  for (i = 0; i < eval_index; ++i) {
261      get_delay(def_ptr, instance, ideat_change,
262          ideat_inconsistent_pins,
263          gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
264  }
265
266  add_inconsistent_pins(instance, ideat_inconsistent_pins);
267
268  clear_ptrns_bits((char *)ideat_change,
269      deb_list(instance->deb_info_index->unit_count);
270
271  clear_ptrns_bits((char *)ideat_inconsistent_pins,
272      deb_list(instance->deb_info_index->unit_count);
273
274  }
275
276  /* Process the STORE pin changes */
277  for (pin_number = instance->first_store_pin_index,
278      instance->first_store_pin_index = -1,
279      pin_number != -1,
280      pin_number = pin_info->next_input_pin_index) {
281      gbl_eval_pin_number[0] = pin_number;
282
283      uwb_ptr = uwb_ptr - offset[pin_number];
284      unitno = uwb_ptr->unitno;
285      wordno = uwb_ptr->wordno;
286      bitno = uwb_ptr->bitno;
287
288      pin_info = (instance->pin_info_table[pin_number]);
289      pin_def = (def_ptr->pin_table[pin_number]);
290
291      pin_info->input_pin_is_linked = FALSE;
292      pin_value = pin_info->old_filtered;
293
294      if (pin_def->direction == IN) {
295          /* INPUT STORE change */
296          old_pin_value = read_pin_value(instance->pin_value,
297              unitno, wordno, bitno);
298
299          set_pin_value(instance->pin_value,
300              unitno, wordno, bitno, pin_value);
301
302          switch (pin_def->clk_format) {
303              case DNRZ:
304                  if (input_pin_transition(old_pin_value, pin_value,
305                      pin_info->uninitialized_pin) == NO_TRANSITION) {
306                      /* This is an error because the host should have filtered the
307                       * transition.
308                       * On success thought it is NOT an error because the host only
309                       * filters transitions to exactly the same value.
310                      */
311                      DPRINTF(("No transition on IN STORE DNRZ pin %s", pin_def->pin_name));
312                      continue;
313                  }
314
315                  if (pin_value & (LOGIC_0 | LOGIC_50 | LOGIC_Z0))
316                      reset_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
317                  else
318                      set_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
319
320                  break;
321
322              case R1:
323                  if (input_pin_transition(old_pin_value, pin_value,
324                      pin_info->uninitialized_pin) != RISE_TRANSITION) {
325                      DPRINTF(("No transition on IN STORE R1 pin %s", pin_def->pin_name));
326                      continue;
327                  }
328
329                  /* Disable the R1 clock on the measurement pattern (ptrns_loaded) */
330                  set_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
331
332                  break;
333
334              case R0:
335                  if (input_pin_transition(old_pin_value, pin_value,
336                      pin_info->uninitialized_pin) != FALL_TRANSITION) {
337                      DPRINTF(("No transition on IN STORE R0 pin %s", pin_def->pin_name));
338                      continue;
339                  }
340
341                  /* Disable the RZ clock on the measurement pattern (ptrns_loaded) */
342                  reset_ptrns_bit(instance->ptrns_loaded[unitno], wordno, bitno);
343
344                  break;
345
346              default:
347                  lm_queue_message(ERROR_MSG, "internal error: illegal pin type in device.b");
348                  continue;
349          }
350      }
351
352      else {
353          /* IO STORE change */
354          /* Note: IO store pin can only have DNRZ format */
355          old_pin_value = read_pin_value(instance->last_sample_value,

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE	5/23/89	PAGE #
TIME	6:14:48 pm	4/115

```

LINE # SOURCE TEXT
361         unitno, wordno, bitno);
362
363     set_pin_value(instance->pin_value,
364         unitno, wordno, bitno, pin_value);
365
366     if (io_pin_transition(old_pin_value, pin_value,
367         pin_info->uninitialized_pin) == NO_TRANSITION) {
368
369         DPRINTF(("No transition on IO STORE pin %s", pin_def->pin_name));
370
371         continue;
372     }
373
374     set_measurement_pattern(instance, pin_def,
375         unitno, wordno, bitno, pin_value,
376         &lcycdb_modified, &lcycdb_modified);
377
378     if (lcycdb_modified == TRUE) {
379         write_pattern(instance,
380             instance->lcycdb_addr[0],
381             instance->lcycdb_loaded);
382         lcycdb_modified = FALSE;
383     }
384
385     if (lcycdb_modified == TRUE) {
386         write_pattern(instance,
387             instance->lcycdb_addr[0],
388             instance->lcycdb_loaded);
389         lcycdb_modified = FALSE;
390     }
391 }
392
393 calculate_consistent_set(instance, def_ptr, consistent_set);
394
395 switch (pin_def->clk_format) {
396 case R1:
397     reset_ptrn_bit(&consistent_set[unitno], wordno, bitno);
398     break;
399 case R0:
400     set_ptrn_bit (&consistent_set[unitno], wordno, bitno);
401     break;
402 default:
403     break;
404 }
405
406 /* save the consistent set */
407 copy_ptrn_bits((char *)consistent_set,
408     (char *)instance->last_consistent_set,
409     dab_list(instance->dab_info_index->unit_count);
410
411 if (instance->use_2_bit_per_pin == TRUE) {
412     switch_to_2_bit_per_pin(instance);
413 }
414
415 write_pattern(instance,
416     instance->unit_addr(instance->cur_unit_addr_index)[0],
417     consistent_set);
418
419 if (grow_pattern(instance) == FAILURE)
420     return(FAILURE);
421
422 write_pattern(instance,
423     instance->unit_addr(instance->cur_unit_addr_index)[0],
424     instance->ptrn_loaded);
425
426 set_seq_end_bit(instance,
427     instance->last_addr,
428     FALSE,
429     seq_end_addr,
430     inst_block_number);
431
432 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
433     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
434     return(FAILURE);
435 }
436
437 if (get_result(def_ptr, instance, ident_change,
438     STORE_EVENT, gbl_eval_pin_number,
439     1, any_driving_to_z) == FAILURE) {
440     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
441     return(FAILURE);
442 }
443
444 for (i = 0; i < replay_count; ++i) {
445     if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
446         remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
447         return(FAILURE);
448     }
449
450     read_magic_full_sample_reg(instance, temp_steady_state_result);
451     check_consistency(instance,
452         instance->last_sample_value,
453         temp_steady_state_result,
454         ident_inconsistent_pins,
455         ident_change,
456         total_unit);
457 }
458
459 #ifdef DEBUG
460 if (loop_till_key == TRUE) {
461     printf("looping in eval instance. STORE change...\n");
462     debug_key = 0;
463     while (debug_key == 0) {
464         if (play_ptrn_seq(instance,
465             timeout, &changed_dac) == FAILURE) {
466             remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
467             return(FAILURE);
468         }
469     }
470
471     if (debug_char == (u_long)'t') {
472         loop_till_key = FALSE;
473         printf("stop loop\n");
474     }
475     debug_key = 0;
476 }
477 #endif
478
479 if (any_driving_to_z == TRUE) {

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89 PAGE #  
TIME 6:14:48 pm 5/116

```

LINE # SOURCE TEXT
481 /* Use 2 bits per pin -> hard drive those IO STUFF pins which
482 change from driving to Z.
483 */
484
485 if (def_ptr->device_type == PUBLIC) {
486     /* Write memory LOADED to previous pattern addr...
487     * For PRIVATE devices we don't have to do anything since
488     * ROMS will always be written when we do the next evaluation.
489     */
490     write_pattern(instance,
491         (instance->unit_addr[
492             instance->cur_unit_addr_index + 1 & 1][0],
493             instance->hndmb_loaded);
494
495     write_pattern(instance,
496         (instance->unit_addr[instance->cur_unit_addr_index][0],
497             instance->last_consistent_set);
498
499     if (grow_pattern(instance) == FAILURE)
500         return(FAILURE);
501 }
502
503 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
504
505 get_delay(def_ptr, instance, ident_change,
506     ident_inconsistent_pin,
507     (u_short)pin_number, pin_value);
508
509 add_inconsistent_pins(instance, ident_inconsistent_pin);
510
511 clear_ptrn_bits((char *)ident_change,
512     dab_list(instance->dab_info_index)->unit_count);
513
514 clear_ptrn_bits((char *)ident_inconsistent_pin,
515     dab_list(instance->dab_info_index)->unit_count);
516
517 pin_info->uninitialized_pin = FALSE;
518
519 }
520
521 return(SUCCESS);
522 }
523
524 set_measurement_pattern(instance, pin_def, unitno, wordno, bitno, pin_value,
525     lcyddb_modified, lcyddb_modified);
526
527 INSTANCE INFO
528 PIN_SPEC
529 u_char
530 u_char
531 u_char
532 u_char
533 u_char
534 {
535     u_char measured_value;
536
537     if (pin_value & (LOGIC_0 | LOGIC_S0))
538         reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
539     else if (pin_value & (LOGIC_1 | LOGIC_S1))
540         set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
541     else {
542         measured_value = read_pin_value(instance->last_sample_value,
543             unitno, wordno, bitno);
544         if (measured_value & (LOGIC_0 | LOGIC_S0))
545             reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
546         else if (measured_value & (LOGIC_1 | LOGIC_S1))
547             set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
548         else {
549             /* Simulator value is I/O and measured value is U...Drive
550             * this pin to the opposite value.
551             */
552             toggle_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
553         }
554     }
555
556     if (instance->definition->device_type == PRIVATE) {
557         if (pin_value & (LOGIC_0 | LOGIC_S0 | LOGIC_S1)) {
558             reset_ptrn_bit(instance->hndmb_loaded[unitno], wordno, bitno);
559         }
560         else {
561             set_ptrn_bit(instance->hndmb_loaded[unitno], wordno, bitno);
562         }
563         return;
564     }
565
566     if (pin_def->direction == IO) {
567         if (pin_value & (LOGIC_0 | LOGIC_1 | LOGIC_S0 | LOGIC_S1)) {
568             switch (pin_def->last_cyc_drive) {
569                 case R_DRIVE:
570                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
571                     *lcyddb_modified = TRUE;
572                     break;
573                 case M_DRIVE:
574                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
575                     *lcyddb_modified = TRUE;
576                     break;
577                 case W_DRIVE:
578                     reset_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
579                     *lcyddb_modified = TRUE;
580                     *lcyddb_modified = TRUE;
581                     break;
582                 case NO_DRIVE:
583                     break;
584                 default:
585                     break;
586             }
587         }
588         else {
589             switch (pin_def->last_cyc_drive) {
590                 case R_DRIVE:
591                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
592                     *lcyddb_modified = TRUE;
593                     break;
594                 case M_DRIVE:
595                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
596                     *lcyddb_modified = TRUE;
597                     break;
598                 case W_DRIVE:
599                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);
600                     set_ptrn_bit(instance->lcyddb_loaded[unitno], wordno, bitno);

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 07/23/89

PAGE #

TIME 6:14:48 pm

6/117

LINE # SOURCE TEXT

```

601  *leycade_modified = TRUE;
602  *leycade_modified = TRUE;
603  break;
604  case NO_DRIVE:
605  default:
606  break;
607  }
608  }
609  }
610  }
611  }
612  calculate_consistent_set(instance, def_ptr, consistent_set_ptr)
613  {
614  INSTANCE_INFO *instance;
615  DEVICE_SPEC *def_ptr;
616  PTRN_BITS_LONGWORD *consistent_set_ptr;
617  {
618  EXTRA_DEVICE_SPEC *extra_def_ptr;
619  DAB_INFO *dab_ptr;
620  PTRN_BITS_LONGWORD *ptrn_loaded_ptr;
621  PTRN_BITS_LONGWORD *sampled_data_ptr;
622  PTRN_BITS_LONGWORD *sampled_hiz_ptr;
623  PTRN_BITS_LONGWORD *sampled_unk_ptr;
624  PTRN_BITS_LONGWORD *sim_data_ptr;
625  PTRN_BITS_LONGWORD *sim_hiz_ptr;
626  PTRN_BITS_LONGWORD *sim_unk_ptr;
627  PTRN_BITS_LONGWORD *ident_inputs_ptr;
628  PTRN_BITS_LONGWORD *ident_outputs_ptr;
629  PTRN_BITS_LONGWORD *ident_los_ptr;
630  u_char total_unit;
631  u_char unit;
632  u_char word;
633  }
634  /* Calculate consistent set:
635  * for INPUT pins --> take the sim_pin_value (or ptrn_loaded)
636  *   disable R1/R2 clocks
637  * for OUTPUT pins --> take the last_sample_value
638  * for IO pins --> take the combination of sim_pin_value and
639  *   last_sample_value.
640  */
641  extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
642  dab_ptr = dab_list(instance->dab_info_index);
643  total_unit = dab_ptr->unit_count;
644  ptrn_loaded_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded;
645  sampled_data_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.data;
646  sampled_hiz_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz;
647  sampled_unk_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.unknown;
648  sim_data_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.data;
649  sim_hiz_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz;
650  sim_unk_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.unknown;
651  ident_inputs_ptr = extra_def_ptr->ident_inputs;
652  ident_outputs_ptr = extra_def_ptr->ident_outputs;
653  ident_los_ptr = extra_def_ptr->ident_los;
654  for (unit = 0; unit < total_unit; ++unit) {
655  for (word = 0; word < J; ++word) {
656  /* INPUT pins.
657  * The ptrn_loaded already contains the clock and the data bits
658  * set correctly before this step, so just copy it. We cannot
659  * use the sim_pin_value because the clock (R1/R2) might not be
660  * set correctly. For example if a pin is an R1 pin and we have
661  * seen the falling transition but not the rising transition yet.
662  * At that point the sim_pin_value will be set to 0. If some
663  * other store pin change value and we use sim_pin_value to
664  * build the consistent set, then the consistent set will have
665  * that clock enabled (which is wrong !!!).
666  */
667  consistent_set_ptr->word[word] =
668  ptrn_loaded_ptr->word[word] & ident_inputs_ptr->word[word];
669  /* OUTPUT pins.
670  * Drive the output pins to the last sampled value.
671  */
672  consistent_set_ptr->word[word] |=
673  sampled_data_ptr->word[word] & ident_outputs_ptr->word[word];
674  /* IO pins.
675  * Drive the IO pins to the combination of sim_pin_value and
676  * last_sample_value as follows:
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *

```

SIM	ANY	Z	U	ANY
SAMPLE	ANY	Z	U	
RESULT	\$1	\$2	\$3	\$4

```

701  *
702  * $1 --> use sample data
703  * $2 --> use sample data
704  * $3 --> use sim data
705  * $4 --> use the opposite value of sample data
706  * $5 --> use sample data
707  */
708  consistent_set_ptr->word[word] |= ident_los_ptr->word[word] &
709  {
710  /* Result $2 */
711  ((sampled_hiz_ptr->word[word] & sim_hiz_ptr->word[word]) &
712  sampled_data_ptr->word[word]) |
713  /* Result $5 */
714  ((sampled_hiz_ptr->word[word] & sim_unk_ptr->word[word]) &
715  sampled_data_ptr->word[word]) |
716  /* Result $3 */
717  ((sampled_hiz_ptr->word[word] &
718  (sim_hiz_ptr->word[word] | sim_unk_ptr->word[word])) &
719  sim_data_ptr->word[word]) |
720  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
7/118

```

LINE # SOURCE TEXT
721 /* Result #4 */
722 (sampled_unk_ptr->word[word] & ~ sampled_data_ptr->word[word]) |
723
724 /* Result #1 */
725 (~ (sampled_hiz_ptr->word[word] | sampled_unk_ptr->word[word]) &
726   sampled_data_ptr->word[word])
727
728 }
729
730 }
731
732 /* copy the control bits */
733 ((PTRN_BITS ^ consistent_set_ptr)->ctl =
734   ((PTRN_BITS ^ ptrn_loaded_ptr)->ctl,
735
736   ++ptrn_loaded_ptr;
737   ++sampled_data_ptr;
738   ++sampled_hiz_ptr;
739   ++sampled_unk_ptr;
740   ++sin_data_ptr;
741   ++sin_hiz_ptr;
742   ++sin_unk_ptr;
743   ++ident_inputs_ptr;
744   ++ident_outputs_ptr;
745   ++ident_ios_ptr;
746   ++consistent_set_ptr;
747 }
748
749
750 get_result(def_ptr, instance, ident_change,
751   event_type, eval_pin_number, eval_pin_count, any_driving_to_z)
752
753 DEVICE_SPEC *def_ptr;
754 INSTANCE_INFO *instance;
755 PTRN_BITS_LONGWORD *ident_change;
756 u_char event_type;
757 u_short eval_pin_number;
758 u_short eval_pin_count;
759 u_char any_driving_to_z;
760
761 /* Read the result from the magic chip and set the variable "ident_change".
762 * "event_type" determines if it is legal to have IO STORE transitions.
763 * "eval_pin_number" and "eval_pin_count" are used to report errors;
764 * and ONLY used if "event_type" is EVAL_EVENT.
765 * Also change the output pin's instance->ptrn_loaded to be the new
766 * sample value.
767 */
768
769 DAB_INFO *dab_ptr;
770 EXTRA_DEVICE_SPEC *extra_def_ptr;
771 PIN_SPEC *pin_spec;
772 u_long *new_sample_unk_ptr;
773 u_long *new_sample_hiz_ptr;
774 u_long *new_sample_data_ptr;
775 u_long *last_sample_unk_ptr;
776 u_long *last_sample_hiz_ptr;
777 u_long *last_sample_data_ptr;
778 u_long *ptrn_loaded_ptr;
779 u_long *ident_ios_ptr;
780 u_long *ident_outputs_ptr;
781 u_long *ident_change_ptr;
782 u_long *ident_store_ptr;
783 u_long *ident_driving_to_z;
784 u_long *ident_x_to_driving;
785 u_long *ident_io_store;
786 u_long *mask;
787 u_long *ident_hiz_io_change;
788 u_short pin_number;
789 char pin_name_list[MAX_PIN_LENGTH];
790 u_char build_pin_name_list = FALSE;
791 u_char i;
792 u_char total_word;
793 u_char total_unit;
794 u_char waitno;
795 u_char wordno;
796 u_char bitno;
797 u_char unit_offset;
798 u_char word_offset;
799 u_short word_pin_number_offset;
800 u_short unit_pin_number_offset;
801
802 DPRINTF(("inside get_result\n"));
803
804 any_driving_to_z = FALSE;
805
806 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
807 dab_ptr = dab_list(instance->dab_info_index);
808 total_word = dab_ptr->unit_count *
809   sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
810
811 read_magic_full_sample_reg(instance, &gbl_new_sample_value);
812
813 new_sample_unk_ptr = (u_long *)gbl_new_sample_value.unknown;
814 new_sample_hiz_ptr = (u_long *)gbl_new_sample_value.hiz;
815 new_sample_data_ptr = (u_long *)gbl_new_sample_value.data;
816
817 last_sample_unk_ptr = (u_long *)instance->last_sample_value.unknown;
818 last_sample_hiz_ptr = (u_long *)instance->last_sample_value.hiz;
819 last_sample_data_ptr = (u_long *)instance->last_sample_value.data;
820
821 ptrn_loaded_ptr = (u_long *)instance->ptrn_loaded;
822
823 ident_ios_ptr = (u_long *)extra_def_ptr->ident_ios;
824 ident_outputs_ptr = (u_long *)extra_def_ptr->ident_outputs;
825 ident_store_ptr = (u_long *)extra_def_ptr->ident_store;
826 ident_change_ptr = (u_long *)ident_change;
827
828 if (instance->first_eval == TRUE) {
829   DPRINTF(("first eval\n"));
830   for (wordno = 0; wordno < total_word; ++wordno) {
831     ident_change_ptr[wordno] |= 0xffffffff;
832
833     ident_change_ptr[wordno] |=
834       ident_ios_ptr[wordno] | ident_outputs_ptr[wordno];
835
836     ptrn_loaded_ptr[wordno] =
837       (ptrn_loaded_ptr[wordno] & ~ ident_ios_ptr[wordno]) |
838       (new_sample_data_ptr[wordno] & ident_ios_ptr[wordno]);
839
840 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

\*

DATE	5/23/89	PAGE #
TIME	6:14:48 pm	8/119

```

LINE # SOURCE TEXT
841 instance->first_word = FALSE;
842 }
843 else {
844 {
845 instance->first_11 = FALSE;
846 for (wordno = 0; wordno < total_word; ++wordno) {
847 /* Ignore the changes from 11 to 10 and vice versa */
848 ident_hiz_no_change =
849 new_sample_hiz_ptr[wordno] & last_sample_hiz_ptr[wordno];
850
851 ident_change_ptr[wordno] |=
852 new_sample_hiz_ptr[wordno] ^ last_sample_hiz_ptr[wordno] |
853 new_sample_hiz_ptr[wordno] ^ last_sample_hiz_ptr[wordno] |
854 (new_sample_data_ptr[wordno] ^ last_sample_data_ptr[wordno]) &
855 ident_hiz_no_change;
856
857 /* We are interested only in IO and OUTPUT changes */
858 ident_change_ptr[wordno] |=
859 ident_ios_ptr[wordno] | ident_outputs_ptr[wordno];
860
861 DPRINTF(("chg word %d: %08x\n", wordno, ident_change_ptr[wordno]));
862
863 ptrs_loaded_ptr[wordno] =
864 (ptrs_loaded_ptr[wordno] & "ident_ios_ptr[wordno] |
865 new_sample_data_ptr[wordno] & ident_ios_ptr[wordno]);
866 }
867 }
868
869 if (extra_def_ptr->has_io_store == TRUE) {
870
871 total_unit = def_ptr->unit_count;
872
873 /* Check if any IO STORE pins go from driving to 2 */
874 unit_offset = 0;
875 unit_ptr = 0;
876 for (unitno = 0; unitno < total_unit; ++unitno) {
877
878 word_pin_number_offset = unit_pin_number_offset + 79;
879
880 for (wordno = 0; wordno < 3; ++wordno) {
881
882 word_offset = unit_offset + wordno;
883
884 ident_io_store = ident_ios_ptr[word_offset] &
885 ident_store_ptr[word_offset];
886
887 /*
888 ptrs_loaded_ptr[word_offset] =
889 (ptrs_loaded_ptr[word_offset] & "ident_io_store |
890 new_sample_data_ptr[word_offset] & ident_io_store);
891
892 if (ident_driving_to_2 =
893 (ident_io_store) &
894 (last_sample_hiz_ptr[word_offset] &
895 new_sample_hiz_ptr[word_offset])) {
896
897 *any_driving_to_2 = TRUE;
898
899 for (bitno = 31; bitno >= 0; --bitno) {
900 mask = bitno_to_mask(bitno);
901
902 if (ident_driving_to_2 == 0)
903 break;
904
905 if (ident_driving_to_2 & mask) {
906 ident_driving_to_2 = mask;
907
908 if ((event_type == EVAL_EVENT) &&
909 (def_ptr->report_ioac == TRUE)) {
910 if (build_pin_name_list == FALSE) {
911
912 /* Get a list of EVAL pin names */
913 pin_name_list[0] = '\0';
914
915 for (i = 0; i < eval_pin_count; ++i) {
916 pin_spec = def_ptr->
917 pin_table[eval_pin_number[i]];
918
919 if (strlen(pin_name_list) +
920 strlen(pin_spec->pin_name) + 1 <
921 MAX_LINE_LENGTH) {
922 (void)strcat(pin_name_list,
923 pin_spec->pin_name);
924 (void)strcat(pin_name_list, " ");
925 }
926 else
927 break;
928 }
929
930 build_pin_name_list = TRUE;
931 }
932
933 pin_number = word_pin_number_offset + bitno - 31;
934
935 in_queue_message(WARNING_MSG, "any of these eval pins: %s of instance: %s caused I/O store pin: %s to change from driv-
936 ing to 2");
937 pin_name_list,
938 instance->device_info_string,
939 def_ptr->pin_table[pin_number].pin_name);
940
941 reset_ptr_bit_low(
942 (PTRN_BITS_LONGWORD *)instance->hmdab_loaded[unitno],
943 wordno, (u_char)bitno);
944 }
945 }
946
947 word_pin_number_offset += 32;
948 }
949
950 unit_offset += sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
951 unit_pin_number_offset += 80;
952 }
953
954 /* Check if any IO STORE pins go from 2 to driving */
955 unit_pin_number_offset = 0;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
9/120

```

LINE # SOURCE TEXT
960 unit_offset = 0;
961
962 for (unitno = 0; unitno < total_unit; ++unitno) {
963
964     word_pin_number_offset = unit_pin_number_offset + 79;
965
966     for (wordno = 0; wordno < 3; ++wordno) {
967
968         word_offset = unit_offset + wordno;
969
970         if (ident_x_to_driving =
971             (ident_ios_ptr[word_offset] & ident_store_ptr[word_offset]) &
972             (last_sample_hiz_ptr[word_offset] &
973              new_sample_hiz_ptr[word_offset])) {
974
975             instance->use_2_bit_per_pin = TRUE;
976
977             for (bitno = 31; bitno >= 0; --bitno) {
978                 mask = bitno_to_mask(bitno);
979
980                 if (ident_x_to_driving == 0)
981                     break;
982
983                 if (ident_x_to_driving & mask) {
984                     ident_x_to_driving = mask;
985
986                     if ((event_type == EVAL_EVENT) &&
987                         (def_ptr->report_ios == TRUE)) {
988                         if (build_pin_name_list == FALSE) {
989
990                             /* Get a list of EVAL pin names */
991                             pin_name_list[0] = '\0';
992
993                             for (i = 0; i < eval_pin_count; ++i) {
994                                 pin_spec = &def_ptr->
995                                     pin_table[eval_pin_number[i]];
996
997                                 if (strlen(pin_name_list) +
998                                     strlen(pin_spec->pin_name) <
999                                     MAX_LINE_LENGTH) {
1000                                     (void)strcat(pin_name_list,
1001                                         pin_spec->pin_name);
1002                                     (void)strcat(pin_name_list, " ");
1003                                 }
1004                                 else
1005                                     break;
1006                             }
1007
1008                             build_pin_name_list = TRUE;
1009                         }
1010
1011                         pin_number = word_pin_number_offset + bitno - 31;
1012
1013                         in_queue_message(WARNING_MSG, "any of these eval pins: %s of instance: %s caused I/O store pin: %s to change from 2 to
1014 driver",
1015 pin_name_list,
1016 instance->device_info_string,
1017 def_ptr->pin_table[pin_number].pin_name);
1018
1019 set_ptr->bit_long(
1020 PTRN_BITS_LONGWORD *)&instance->hndenb_loaded[unitno],
1021 wordno, (u_char)bitno);
1022
1023 }
1024
1025 }
1026
1027 word_pin_number_offset += 32;
1028
1029 unit_offset += sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
1030 unit_pin_number_offset += 80;
1031 }
1032
1033 /* Save new_sample_values to instance->last_sample_value */
1034 copy_ptrn_bits((char *)new_sample_ios_ptr,
1035 (char *)last_sample_ios_ptr,
1036 dah_ptr->unit_count);
1037 copy_ptrn_bits((char *)new_sample_hiz_ptr,
1038 (char *)last_sample_hiz_ptr,
1039 dah_ptr->unit_count);
1040 copy_ptrn_bits((char *)new_sample_data_ptr,
1041 (char *)last_sample_data_ptr,
1042 dah_ptr->unit_count);
1043
1044 return(SUCCESS);
1045 }
1046
1047 check_consistency(instance, sample_result1, sample_result2,
1048 ident_inconsistent_pins, ident_change, total_unit)
1049
1050 INSTANCE_INFO instance;
1051 FULL_VALUE sample_result1;
1052 FULL_VALUE sample_result2;
1053 PTRN_BITS_LONGWORD ident_inconsistent_pins;
1054 PTRN_BITS_LONGWORD ident_change;
1055 u_char total_unit;
1056 {
1057     /* Compare sample_result1 and sample_result2.
1058     * ADD the differences to ident_inconsistent_pins and ident_change.
1059     */
1060     EXTRA_DEVICE_SPEC extra_def_ptr;
1061     PTRN_BITS_LONGWORD s1_data_ptr;
1062     PTRN_BITS_LONGWORD s1_hiz_ptr;
1063     PTRN_BITS_LONGWORD s1_unknown_ptr;
1064     PTRN_BITS_LONGWORD s2_data_ptr;
1065     PTRN_BITS_LONGWORD s2_hiz_ptr;
1066     PTRN_BITS_LONGWORD s2_unknown_ptr;
1067     PTRN_BITS_LONGWORD ident_outputs_ptr;
1068     PTRN_BITS_LONGWORD ident_ios_ptr;
1069     u_char unitno;
1070     u_char wordno;
1071     #ifdef DEBUG
1072     inconsistent_pins;
1073     mask;
1074     u_long;
1075     u_short;
1076     u_char;
1077     char;
1078     bitno;
1079     #endif

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/ptrnhist.c	DATE 5/23/89	PAGE # 10/121
LINE #	SOURCE TEXT			
1079	extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data,			
1080	ident_outputs_ptr = extra_def_ptr->ident_outputs,			
1081	ident_ios_ptr = extra_def_ptr->ident_ios,			
1082				
1083	s1_data_ptr = (PTRN_BITS_LONGWORD *)sample_result1->data,			
1084	s1_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result1->hiz,			
1085	s1_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result1->unknown,			
1086				
1087	s2_data_ptr = (PTRN_BITS_LONGWORD *)sample_result2->data,			
1088	s2_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result2->hiz,			
1089	s2_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result2->unknown,			
1090				
1091	for (unitno = 0; unitno < total_unit; ++unitno) {			
1092	for (wordno = 0; wordno < 3; ++wordno) {			
1093				
1094	ident_inconsistent_ptr->word[wordno]  =			
1095	(s1_data_ptr->word[wordno] ^ s2_data_ptr->word[wordno])			
1096	s1_hiz_ptr->word[wordno] ^ s2_hiz_ptr->word[wordno]			
1097	s1_unknown_ptr->word[wordno] ^ s2_unknown_ptr->word[wordno]) &			
1098	(ident_outputs_ptr->word[wordno]			
1099	ident_ios_ptr->word[wordno]),			
1100				
1101	ident_change->word[wordno]  =			
1102	ident_inconsistent_ptr->word[wordno],			
1103				
1104	#ifdef DEBUG			
1105	if (ident_inconsistent_ptr->word[wordno]) {			
1106	inconsistent_ptr = ident_inconsistent_ptr->word[wordno],			
1107				
1108	for (bitno = 31; bitno >= 0; --bitno) {			
1109	if (inconsistent_ptr->bitno == 0)			
1110	break;			
1111	mask = bitno < 31 ? mask[bitno] :			
1112	if (inconsistent_ptr->mask & mask) {			
1113	inconsistent_ptr->mask = mask;			
1114				
1115	pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);			
1116	DPRINTF(("PIN: %d is found inconsistent\n", pin_number));			
1117				
1118	}			
1119	any_inconsistent_ptr = TRUE;			
1120	}			
1121	#endif			
1122				
1123				
1124				
1125				
1126	++ident_inconsistent_ptr,			
1127	++ident_outputs_ptr,			
1128	++ident_ios_ptr,			
1129	++ident_change,			
1130	++s1_data_ptr,			
1131	++s1_hiz_ptr,			
1132	++s1_unknown_ptr,			
1133	++s2_data_ptr,			
1134	++s2_hiz_ptr,			
1135	++s2_unknown_ptr,			
1136				
1137	#ifdef DEBUG			
1138				
1139	if (any_inconsistent_ptr == TRUE) {			
1140	DPRINTF(("there are inconsistent pins\n"));			
1141				
1142	#endif			
1143				
1144				
1145	check_consistency2(instance, sample_result1, sample_result2,			
1146	ident_inconsistent_ptr, ident_change, total_unit)			
1147	INSTANCE INFO			
1148	FULL_VALUE			
1149	FULL_VALUE			
1150	PTRN_BITS_LONGWORD			
1151	PTRN_BITS_LONGWORD			
1152	u_char			
1153				
1154	/* Compare sample_result1 and sample_result2.			
1155	* Add the difference to ident_inconsistent_ptr and ident_change.			
1156	* Don't compare data if it is RIZ or UNKNOWN.			
1157	*/			
1158	EXTRA_DEVICE_SPEC			
1159	PTRN_BITS_LONGWORD			
1160	PTRN_BITS_LONGWORD			
1161	PTRN_BITS_LONGWORD			
1162	PTRN_BITS_LONGWORD			
1163	PTRN_BITS_LONGWORD			
1164	PTRN_BITS_LONGWORD			
1165	PTRN_BITS_LONGWORD			
1166	PTRN_BITS_LONGWORD			
1167	u_long			
1168	u_char			
1169	u_char			
1170	#ifdef DEBUG			
1171	u_long			
1172	u_long			
1173	u_short			
1174	u_char			
1175	char			
1176	#endif			
1177				
1178	extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data,			
1179	ident_outputs_ptr = extra_def_ptr->ident_outputs,			
1180	ident_ios_ptr = extra_def_ptr->ident_ios,			
1181				
1182	s1_data_ptr = (PTRN_BITS_LONGWORD *)sample_result1->data,			
1183	s1_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result1->hiz,			
1184	s1_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result1->unknown,			
1185				
1186	s2_data_ptr = (PTRN_BITS_LONGWORD *)sample_result2->data,			
1187	s2_hiz_ptr = (PTRN_BITS_LONGWORD *)sample_result2->hiz,			
1188	s2_unknown_ptr = (PTRN_BITS_LONGWORD *)sample_result2->unknown,			
1189				
1190	for (unitno = 0; unitno < total_unit; ++unitno) {			
1191	for (wordno = 0; wordno < 3; ++wordno) {			
1192				
1193	ident_hiz_and_unk =			
1194	s1_hiz_ptr->word[wordno]			
1195	s2_hiz_ptr->word[wordno]			
1196	s1_unknown_ptr->word[wordno]			
1197	s2_unknown_ptr->word[wordno]),			
1198				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
11/122

```

1199 ident_inconsistent_pins->word[wordno] |=
1200 (((s1_data_ptr ->word[wordno]) & s2_data_ptr ->word[wordno]) &
1201  ident_hiz_4sd_unk) |
1202  s1_hiz_ptr ->word[wordno] & s2_hiz_ptr ->word[wordno] |
1203  s1_unk_ptr ->word[wordno] & s2_unk_ptr ->word[wordno] &
1204  s2_data_ptr ->word[wordno] |
1205  ident_ios_ptr ->word[wordno]);
1206
1207 ident_change->word[wordno] |=
1208  ident_inconsistent_pins->word[wordno];
1209
1210 #ifdef DEBUG
1211 if (ident_inconsistent_pins->word[wordno]) {
1212   inconsistent_pins = ident_inconsistent_pins->word[wordno];
1213   for (bitno = 31; bitno >= 0; --bitno) {
1214     if (inconsistent_pins &= 0)
1215       break;
1216     mask = bitno <= mask[bitno];
1217     if (inconsistent_pins & mask) {
1218       inconsistent_pins = mask;
1219     }
1220     pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
1221     DPRINTF(("PN: %d is found inconsistent\n", pin_number));
1222   }
1223   any_inconsistent_pins = TRUE;
1224 }
1225 #endif
1226
1227 ++ident_inconsistent_pins;
1228 ++ident_outputs_ptr;
1229 ++ident_ios_ptr;
1230 ++ident_change;
1231 ++s1_data_ptr;
1232 ++s1_hiz_ptr;
1233 ++s1_unk_ptr;
1234 ++s2_data_ptr;
1235 ++s2_hiz_ptr;
1236 ++s2_unk_ptr;
1237
1238 #ifdef DEBUG
1239 if (any_inconsistent_pins == TRUE) {
1240   DPRINTF(("there are inconsistent pins\n"));
1241 }
1242 #endif
1243
1244 copy_in_pin_changes(instance)
1245 INSTANCE_INFO *instance;
1246 {
1247   PIN_INFO *pin_info;
1248   u_short data_pin_count;
1249   u_short eval_pin_count;
1250   u_short store_pin_count;
1251   u_short i;
1252   u_short pin_number;
1253
1254   data_pin_count = lm_get_short();
1255   eval_pin_count = lm_get_short();
1256   store_pin_count = lm_get_short();
1257
1258   for (i = 0; i < data_pin_count; ++i) {
1259     pin_number = lm_get_short();
1260     pin_info = instance->pin_info_table[pin_number];
1261
1262     if (pin_info->input_pin_is_linked == TRUE) {
1263       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1264         pin_number);
1265       return(FAILURE);
1266     }
1267     else {
1268       pin_info->pin_time = lm_get_int();
1269       pin_info->new_raw = lm_get_char();
1270
1271       /* Map IO/EI to EI for pullup OR IO/EI to EO for pulldown. */
1272       MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1273         pin_info->new_raw);
1274       pin_info->input_pin_is_linked = TRUE;
1275       pin_info->next_input_pin_index = instance->first_data_pin_index;
1276       instance->first_data_pin_index = pin_number;
1277     }
1278   }
1279
1280   for (i = 0; i < eval_pin_count; ++i) {
1281     pin_number = lm_get_short();
1282     pin_info = instance->pin_info_table[pin_number];
1283
1284     if (pin_info->input_pin_is_linked == TRUE) {
1285       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1286         pin_number);
1287       return(FAILURE);
1288     }
1289     else {
1290       pin_info->pin_time = lm_get_int();
1291       pin_info->new_raw = lm_get_char();
1292
1293       /* Map IO/EI to EI for pullup OR IO/EI to EO for pulldown. */
1294       MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1295         pin_info->new_raw);
1296       pin_info->input_pin_is_linked = TRUE;
1297       pin_info->next_input_pin_index = instance->first_eval_pin_index;
1298       instance->first_eval_pin_index = pin_number;
1299     }
1300   }
1301
1302   for (i = 0; i < store_pin_count; ++i) {
1303     pin_number = lm_get_short();
1304     pin_info = instance->pin_info_table[pin_number];
1305
1306     if (pin_info->input_pin_is_linked == TRUE) {
1307       lm_queue_message(ERROR_MSG, "internal error: duplicate pin info from HOST (pinno: %d)",
1308         pin_number);
1309       return(FAILURE);
1310     }
1311     else {
1312

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/ptrnhist.c

DATE

5/23/89

PAGE #

12/123

TIME

6:14:48 pm

```

1319     pin_info->pin_time = lm_get_int();
1320     pin_info->new_raw = lm_get_char();
1321
1322     /* Map Z0/Z1 to S1 for pulldown OR Z0/Z1 to S0 for pulldown. */
1323     MAP_IN_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1324         pin_info->new_raw);
1325     pin_info->input_pin_is_linked = TRUE;
1326     pin_info->next_input_pin_index = instance->first_store_pin_index;
1327     instance->first_store_pin_index = pin_number;
1328
1329     }
1330
1331     return(SUCCESS);
1332 }
1333
1334 copy_out_pin_changes(instance)
1335 INSTANCE_INFO *instance;
1336 {
1337     /* Copy the IO and OUTPUT changes from the PIN_INFO_TABLE to the
1338      * network buffer. These changes are linked with first_data_pin_index
1339      * as the head of the list.
1340      */
1341
1342     PIN_INFO *pin_info;
1343     u_long output_count_mark;
1344     short pin_number;
1345     short pin_count = 0;
1346
1347     output_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1348     lm_put_short(0);
1349
1350     pin_number = instance->first_data_pin_index;
1351     instance->first_data_pin_index = -1;
1352
1353     while (pin_number != -1) {
1354         ++pin_count;
1355         pin_info = instance->pin_info_table(pin_number);
1356         pin_info->output_pin_is_linked = FALSE;
1357
1358         DPRINTF(("chg PW: ad val: td EPN: ad ",
1359             pin_number, pin_info->old_filtered,
1360             pin_info->event_pin_number));
1361         lm_put_short(pin_number);
1362
1363         /* Map Z1 to S1 for pulldown OR Z0 to S0 for pulldown. */
1364         MAP_OUT_LOGIC_VALUE(pin_number, pin_info->has_resistor,
1365             pin_info->direction,
1366             pin_info->old_filtered);
1367
1368         lm_put_char(pin_info->old_filtered);
1369         lm_put_short(pin_info->event_pin_number);
1370         lm_put_char(pin_info->delay_type);
1371
1372         switch (pin_info->delay_type) {
1373             default:
1374                 case DELAY_TABLE_COMPOSITE:
1375                     DPRINTF(("td: td: td: ",
1376                         pin_info->min_delay,
1377                         pin_info->typ_delay, pin_info->max_delay));
1378                     lm_put_int(pin_info->min_delay);
1379                     lm_put_int(pin_info->typ_delay);
1380                     lm_put_int(pin_info->max_delay);
1381                     break;
1382                 case DELAY_TABLE:
1383                     DPRINTF(("td: td: ", pin_info->min_delay));
1384                     lm_put_int(pin_info->min_delay);
1385                     break;
1386                 case MEASURED:
1387                     DPRINTF(("meas: td:td: ", pin_info->min_delay, pin_info->max_delay));
1388                     lm_put_int(pin_info->min_delay);
1389                     lm_put_int(pin_info->max_delay);
1390                     break;
1391             }
1392         DPRINTF(("a"));
1393
1394         pin_number = pin_info->next_output_pin_index;
1395     }
1396
1397     LM_PUT_SHORT_AT_MARK(output_count_mark, lm_global_conn_ptr, pin_count);
1398 }
1399
1400 run_input_bcode(instance)
1401 INSTANCE_INFO *instance;
1402 {
1403     /* The B-code uses the pin values in OLD_RAW, OLD_FILTERED, NEW_RAW, and
1404      * NEW_FILTERED. And leave the new modified pin values in OLD_FILTERED.
1405      * It should maintain the pin change link list.
1406      */
1407
1408     /* Since we don't have the B-code yet, just copy the pin values from
1409      * NEW_RAW to OLD_FILTERED.
1410      */
1411
1412     PIN_INFO *pin_info;
1413     short pin_number;
1414
1415     /* Copy the DATA pin changes */
1416     pin_number = instance->first_data_pin_index;
1417
1418     while (pin_number != -1) {
1419         pin_info = instance->pin_info_table(pin_number);
1420         pin_info->old_filtered = pin_info->new_raw;
1421         pin_number = pin_info->next_input_pin_index;
1422     }
1423
1424     /* Copy the EVAL pin changes */
1425     pin_number = instance->first_eval_pin_index;
1426
1427     while (pin_number != -1) {
1428         pin_info = instance->pin_info_table(pin_number);
1429         pin_info->old_filtered = pin_info->new_raw;
1430         pin_number = pin_info->next_input_pin_index;
1431     }
1432
1433     /* Copy the STORE pin changes */
1434     pin_number = instance->first_store_pin_index;
1435
1436     while (pin_number != -1) {
1437         pin_info = instance->pin_info_table(pin_number);
1438         pin_info->old_filtered = pin_info->new_raw;
1439         pin_number = pin_info->next_input_pin_index;
1440     }
1441 }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
13/124

LINE # SOURCE TEXT

```

1439 while (pin_number != -1) {
1440     pin_info = instance->pin_info_table(pin_number);
1441     pin_info->old_filtered = pin_info->new_raw;
1442     pin_number = pin_info->next_input_pin_index;
1443 }
1444
1445 run_output_bcode(instance);
1446 INSTANCE_INFO *instance;
1447
1448 PIN_INFO *pin_info;
1449 short pin_number;
1450
1451 /* Copy the OUTPUT pin changes */
1452 pin_number = instance->first_data_pin_index;
1453 while (pin_number != -1) {
1454     pin_info = instance->pin_info_table(pin_number);
1455     pin_info->old_filtered = pin_info->new_raw;
1456     pin_number = pin_info->next_output_pin_index;
1457 }
1458
1459 copy_initial_values(instance);
1460 INSTANCE_INFO *instance;
1461 {
1462     DEVICE_SPEC *definition;
1463     UWB_OFFSET *uwb_ptr;
1464     PIN_SPEC *pin_ptr;
1465     PIN_INFO *pin_info;
1466     u_long output_pin_count_mark;
1467     u_short output_pin_count;
1468     pinno;
1469     u_short pin_count;
1470     u_char pin_value;
1471     u_char unitno;
1472     u_char wordno;
1473     u_char bitno;
1474
1475     definition = instance->definition;
1476
1477     output_pin_count_mark = LM_MARK_BUFFER(lm_global_conn_ptr);
1478     lm_put_short(0); /* dummy out count */
1479     output_pin_count = 0;
1480     pin_count = definition->pin_cnt;
1481     pin_ptr = definition->pin_table;
1482     for (pinno = 0; pinno < pin_count; ++pinno) {
1483         switch (pin_ptr->direction) {
1484             case OUT:
1485                 case IO:
1486                     pin_info = instance->pin_info_table(pinno);
1487                     lm_put_short(pinno);
1488
1489                     uwb_ptr = uwb_ptr->uwb_offset(pinno);
1490                     unitno = uwb_ptr->unitno;
1491                     wordno = uwb_ptr->wordno;
1492                     bitno = uwb_ptr->bitno;
1493
1494                     pin_value = read_pin_value(instance->last_sample_value,
1495                                             unitno, wordno, bitno);
1496
1497                     /* Map Z1 to Z1 for pullup or Z0 to Z0 for pulldown. */
1498                     MAP_OUT_LOGIC_VALUE(pinno, pin_info->has_resistor,
1499                                         pin_info->direction,
1500                                         pin_value);
1501
1502                     lm_put_char(pin_value);
1503
1504                     DPRINTF(("initial val PN: %d val: %d\n", pinno, pin_value));
1505
1506                     ++output_pin_count;
1507                     break;
1508
1509             default:
1510                 break;
1511         }
1512         ++pin_ptr;
1513     }
1514
1515     LM_PUT_SHORT_AT_MARK(output_pin_count_mark,
1516                         lm_global_conn_ptr, output_pin_count);
1517 }
1518
1519 switch_to_2_bit_per_pin(instance);
1520 INSTANCE_INFO *instance;
1521 {
1522     /* Copy the patterns before the measurement patterns to the measurement
1523      * patterns and write the MEASURE patterns to the measurement patterns
1524      * address - 1.
1525      */
1526     DAB_INFO *dab_ptr;
1527     u_long *source_addr_ptr;
1528     u_long *dest_addr_ptr;
1529     u_long source_addr;
1530     u_long dest_addr;
1531     u_long temp;
1532     u_char unitno;
1533     u_char line;
1534     u_char total_unit;
1535
1536     DPRINTF(("inside switch_to_2_bit_per_pin\n"));
1537
1538     #ifdef DONT_FIXIT
1539     write_patterns(instance,
1540                    instance->unit_addr(instance->cur_unit_addr_index)[0],
1541                    instance->hmem_loaded);
1542     #else
1543     source_addr_ptr = instance->unit_addr[
1544                    instance->cur_unit_addr_index + 1][0];
1545     dest_addr_ptr = instance->unit_addr(instance->cur_unit_addr_index)[0];
1546
1547     dab_ptr = dab_list(instance->dab_info_index);
1548     total_unit = dab_ptr->lase_count - dab_ptr->unit_count_per_lase;
1549     for (unitno = 0; unitno < total_unit; ++unitno) {
1550         source_addr = source_addr_ptr[unitno];
1551     }
1552 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/ptrnhist.c

DATE 5/23/89  
TIME 6:14:48 pm

PAGE #  
14/125

LINE #	SOURCE TEXT
1559	dest_addr = dest_addr_ptr[unitno];
1560	
1561	temp = read_loc_long((u_long *)source_addr);
1562	write_loc_long((u_long *)dest_addr, temp);
1563	
1564	temp = read_loc_long((u_long *)source_addr + LANE_SEGMENT_B_OFFSET);
1565	write_loc_long((u_long *)dest_addr + LANE_SEGMENT_B_OFFSET, temp);
1566	
1567	temp = read_loc_long((u_long *)source_addr + LANE_SEGMENT_C_OFFSET);
1568	write_loc_long((u_long *)dest_addr + LANE_SEGMENT_C_OFFSET, temp);
1569	
1570	}
1571	
1572	write_pattern(instance,
1573	instance->unit_addr(instance->cur_unit_addr_index+1][0],
1574	instance->hmdenb_loaded);
1575	endif
1576	
1577	if (grow_pattern(instance) == FAILURE)
1578	return(FAILURE);
1579	
1580	instance->use_2_bit_per_pis = FALSE;
1581	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/saverest.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
1/126

LINE # SOURCE TEXT

```

1  /* SCOS_ID: saverest.c rev 1.1, 4/24/89 at 07:53:52 */
2
3  #include "device.h"
4  #include "hardware.h"
5  #include "aspram.h"
6  #include "lserver.h"
7  #include "lnetwork.h"
8  #include "network.h"
9
10 send_pattern(user, instance)
11 USER_INFO *user,
12 INSTANCE_INFO *instance,
13 {
14     /* Copy one device pattern to the output buffer.
15     * Return FALSE if there are no more patterns to copy to the buffer,
16     * otherwise return TRUE.
17     */
18
19     PTRN_BITS_LONGWORD *ptrn_ptr;
20     DAB_INFO *dab_ptr;
21     u_long eddr;
22     u_long temp;
23     u_char unit_count;
24     u_char wordno;
25     u_char unitno;
26
27     dab_ptr = dab_list(instance->dab_info_index);
28
29     unit_count = dab_ptr->unit_count;
30
31     switch (user->save_state) {
32     case SENT_RECV_NOTHING:
33
34         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded[0];
35
36         for (unitno = 0; unitno < unit_count; ++unitno) {
37             for (wordno = 0; wordno < 3; ++wordno) {
38                 ls_put_int(ptrn_ptr->word[wordno]);
39             }
40             ++ptrn_ptr;
41         }
42
43         user->save_state = SENT_RECV_PTRN_LOADED;
44         break;
45     case SENT_RECV_PTRN_LOADED:
46
47         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->hmdadb_loaded[0];
48
49         for (unitno = 0; unitno < unit_count; ++unitno) {
50             for (wordno = 0; wordno < 3; ++wordno) {
51                 ls_put_int(ptrn_ptr->word[wordno]);
52             }
53             ++ptrn_ptr;
54         }
55
56         user->save_state = SENT_RECV_HMDADB_LOADED;
57         break;
58     case SENT_RECV_HMDADB_LOADED:
59
60         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcycbdb_loaded[0];
61
62         for (unitno = 0; unitno < unit_count; ++unitno) {
63             for (wordno = 0; wordno < 3; ++wordno) {
64                 ls_put_int(ptrn_ptr->word[wordno]);
65             }
66             ++ptrn_ptr;
67         }
68
69         user->save_state = SENT_RECV_LCYCDB_LOADED;
70         break;
71     case SENT_RECV_LCYCDB_LOADED:
72
73         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcycbdb_loaded[0];
74
75         for (unitno = 0; unitno < unit_count; ++unitno) {
76             for (wordno = 0; wordno < 3; ++wordno) {
77                 ls_put_int(ptrn_ptr->word[wordno]);
78             }
79             ++ptrn_ptr;
80         }
81
82         user->save_state = SENT_RECV_LCYCDB_LOADED;
83         break;
84     case SENT_RECV_LCYCDB_LOADED:
85
86         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_consistent_set[0];
87
88         for (unitno = 0; unitno < unit_count; ++unitno) {
89             for (wordno = 0; wordno < 3; ++wordno) {
90                 ls_put_int(ptrn_ptr->word[wordno]);
91             }
92             ++ptrn_ptr;
93         }
94
95         user->save_state = SENT_RECV_LAST_CONSISTENT_SET;
96         break;
97     case SENT_RECV_LAST_CONSISTENT_SET:
98
99         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.data[0];
100
101         for (unitno = 0; unitno < unit_count; ++unitno) {
102             for (wordno = 0; wordno < 3; ++wordno) {
103                 ls_put_int(ptrn_ptr->word[wordno]);
104             }
105             ++ptrn_ptr;
106         }
107
108         user->save_state = SENT_RECV_SIM_PIN_VALUE_DATA;
109         break;
110     case SENT_RECV_SIM_PIN_VALUE_DATA:
111
112         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz[0];
113
114         for (unitno = 0; unitno < unit_count; ++unitno) {
115             for (wordno = 0; wordno < 3; ++wordno) {
116                 ls_put_int(ptrn_ptr->word[wordno]);
117             }
118             ++ptrn_ptr;
119         }
120

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/saverest.c

DATE

5/23/89

PAGE #

TIME

6:14:50 pm

2/127

LINE #	SOURCE TEXT
121	user->save_state = SENT_RECV_SIM_PIN_VALUE_HIZ;
122	break;
123	case SENT_RECV_SIM_PIN_VALUE_HIZ:
124	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->sim_pin_value.unknown[0];
125	
126	for (unitno = 0; unitno < unit_count; ++unitno) {
127	for (wordno = 0; wordno < 3; ++wordno) {
128	lm_put_int(ptrn_ptr->word[wordno]);
129	
130	++ptrn_ptr;
131	
132	}
133	
134	user->save_state = SENT_RECV_SIM_PIN_VALUE_UNK;
135	break;
136	case SENT_RECV_SIM_PIN_VALUE_UNK:
137	
138	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->sim_pin_value.soft[0];
139	
140	for (unitno = 0; unitno < unit_count; ++unitno) {
141	for (wordno = 0; wordno < 3; ++wordno) {
142	lm_put_int(ptrn_ptr->word[wordno]);
143	
144	++ptrn_ptr;
145	
146	}
147	
148	user->save_state = SENT_RECV_SIM_PIN_VALUE_SOFT;
149	break;
150	case SENT_RECV_SIM_PIN_VALUE_SOFT:
151	
152	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->last_sample_value.data[0];
153	
154	for (unitno = 0; unitno < unit_count; ++unitno) {
155	for (wordno = 0; wordno < 3; ++wordno) {
156	lm_put_int(ptrn_ptr->word[wordno]);
157	
158	++ptrn_ptr;
159	
160	}
161	
162	user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_DATA;
163	break;
164	case SENT_RECV_LAST_SAMPLE_VALUE_DATA:
165	
166	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->last_sample_value.hiz[0];
167	
168	for (unitno = 0; unitno < unit_count; ++unitno) {
169	for (wordno = 0; wordno < 3; ++wordno) {
170	lm_put_int(ptrn_ptr->word[wordno]);
171	
172	++ptrn_ptr;
173	
174	}
175	
176	user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_HIZ;
177	break;
178	case SENT_RECV_LAST_SAMPLE_VALUE_HIZ:
179	
180	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->last_sample_value.unknown[0];
181	
182	for (unitno = 0; unitno < unit_count; ++unitno) {
183	for (wordno = 0; wordno < 3; ++wordno) {
184	lm_put_int(ptrn_ptr->word[wordno]);
185	
186	++ptrn_ptr;
187	
188	}
189	
190	user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_UNK;
191	break;
192	case SENT_RECV_LAST_SAMPLE_VALUE_UNK:
193	
194	ptrn_ptr = (PTRN_BITS_LONGWORD *)(&instance->last_sample_value.soft[0];
195	
196	for (unitno = 0; unitno < unit_count; ++unitno) {
197	for (wordno = 0; wordno < 3; ++wordno) {
198	lm_put_int(ptrn_ptr->word[wordno]);
199	
200	++ptrn_ptr;
201	
202	}
203	
204	user->save_state = SENT_RECV_LAST_SAMPLE_VALUE_SOFT;
205	break;
206	case SENT_RECV_LAST_SAMPLE_VALUE_SOFT:
207	
208	if (user->pattern_to_send_count == 0) {
209	user->save_state = SENT_RECV_PATTERN;
210	return(FALSE);
211	}
212	
213	for (unitno = 0; unitno < unit_count; ++unitno) {
214	user->last_unit_sent_addr[unitno] =
215	instance->first_user_ptrn_unit_addr[unitno];
216	
217	addr = user->last_unit_sent_addr[unitno];
218	temp = read_loc_long((u_long *)addr);
219	
220	lm_put_int(temp);
221	
222	temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET));
223	
224	lm_r = int(temp);
225	
226	temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET));
227	
228	lm_put_int(temp);
229	
230	}
231	
232	increment_unit_addr(user->last_unit_sent_addr,
233	unit_count, dab_ptr->unit_count_per_lane);
234	
235	user->pattern_to_send_count -= dab_ptr->unit_count_per_lane;
236	
237	user->save_state = SENT_RECV_PATTERN;
238	break;
239	case SENT_RECV_PATTERN:
240	
241	if (user->pattern_to_send_count == 0) {
242	return(FALSE);
243	}
244	
245	for (unitno = 0; unitno < unit_count; ++unitno) {
246	addr = user->last_unit_sent_addr[unitno];

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/saverest.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
3/128

LINE # SOURCE TEXT

```

241     temp = read_loc_long((u_long *)addr);
242
243     lm_put_int(temp);
244
245     temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET));
246
247     lm_put_int(temp);
248
249     temp = read_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET));
250
251     lm_put_int(temp);
252
253     }
254
255     increment_unit_addr(user->last_unit_sent_addr, unit_count,
256                        dab_ptr->unit_count_per_lane);
257
258     user->pattern_to_send_count -- dab_ptr->unit_count_per_lane;
259
260     break;
261
262     default:
263         return(FALSE);
264     }
265
266     return(TRUE);
267 }
268
269 restore_inst_pattern(instance, unit_count)
270 INSTANCE_INFO *instance;
271 u_short unit_count;
272 {
273     DAB_INFO *dab_ptr;
274     PTRN_BITS_LONGWORD *ptrn_ptr;
275     PTRN_BITS temp_ptrn[MAX_UNIT_COUNT];
276     u_char unitno;
277     u_char wordno;
278
279     dab_ptr = dab_list(instance->dab_info_index);
280
281     switch (instance->restore_state) {
282     case SENT_RECV_NOTHING:
283         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->ptrn_loaded[0];
284
285         for (unitno = 0; unitno < unit_count; ++unitno) {
286             for (wordno = 0; wordno < 3; ++wordno) {
287                 ptrn_ptr->word[wordno] = lm_get_int();
288             }
289
290             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
291
292             ++ptrn_ptr;
293         }
294
295         instance->restore_state = SENT_RECV_PTRN_LOADED;
296         break;
297     case SENT_RECV_PTRN_LOADED:
298         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->hndcnb_loaded[0];
299
300         for (unitno = 0; unitno < unit_count; ++unitno) {
301             for (wordno = 0; wordno < 3; ++wordno) {
302                 ptrn_ptr->word[wordno] = lm_get_int();
303             }
304
305             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
306
307             ++ptrn_ptr;
308         }
309
310         instance->restore_state = SENT_RECV_HNDENB_LOADED;
311         break;
312     case SENT_RECV_HNDENB_LOADED:
313         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcychdb_loaded[0];
314
315         for (unitno = 0; unitno < unit_count; ++unitno) {
316             for (wordno = 0; wordno < 3; ++wordno) {
317                 ptrn_ptr->word[wordno] = lm_get_int();
318             }
319
320             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
321
322             ++ptrn_ptr;
323         }
324
325         instance->restore_state = SENT_RECV_LCYCHDB_LOADED;
326         break;
327     case SENT_RECV_LCYCHDB_LOADED:
328         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->lcycmdb_loaded[0];
329
330         for (unitno = 0; unitno < unit_count; ++unitno) {
331             for (wordno = 0; wordno < 3; ++wordno) {
332                 ptrn_ptr->word[wordno] = lm_get_int();
333             }
334
335             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
336
337             ++ptrn_ptr;
338         }
339
340         instance->restore_state = SENT_RECV_LCYCHDB_LOADED;
341         break;
342     case SENT_RECV_LCYCHDB_LOADED:
343         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->last_consistent_set[0];
344
345         for (unitno = 0; unitno < unit_count; ++unitno) {
346             for (wordno = 0; wordno < 3; ++wordno) {
347                 ptrn_ptr->word[wordno] = lm_get_int();
348             }
349
350             fix_pel_ctl_word(ptrn_ptr, unitno, dab_ptr);
351
352             ++ptrn_ptr;
353         }
354
355         instance->restore_state = SENT_RECV_LAST_CONSISTENT_SET;
356         break;
357     case SENT_RECV_LAST_CONSISTENT_SET:
358         ptrn_ptr = (PTRN_BITS_LONGWORD *)instance->sin_pin_value.data[0];
359
360         for (unitno = 0; unitno < unit_count; ++unitno) {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/saverest.c

DATE	5/23/89	PAGE #
TIME	6:14:50 pm	4/129

LINE #	SOURCE TEXT
361	for (wordao = 0; wordao < 3; ++wordao) {
362	ptrs_ptr->word[wordao] = lm_get_int();
363	}
364	++ptrs_ptr;
365	}
366	instance->restore_state = SENT_RECV_SIM_PIN_VALUE_DATA;
367	break;
368	case SENT_RECV_SIM_PIN_VALUE_DATA:
369	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.hiz[0];
370	for (unitao = 0; unitao < unit_count; ++unitao) {
371	for (wordao = 0; wordao < 3; ++wordao) {
372	ptrs_ptr->word[wordao] = lm_get_int();
373	}
374	++ptrs_ptr;
375	}
376	instance->restore_state = SENT_RECV_SIM_PIN_VALUE_HIZ;
377	break;
378	case SENT_RECV_SIM_PIN_VALUE_HIZ:
379	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.unknown[0];
380	for (unitao = 0; unitao < unit_count; ++unitao) {
381	for (wordao = 0; wordao < 3; ++wordao) {
382	ptrs_ptr->word[wordao] = lm_get_int();
383	}
384	++ptrs_ptr;
385	}
386	instance->restore_state = SENT_RECV_SIM_PIN_VALUE_UNK;
387	break;
388	case SENT_RECV_SIM_PIN_VALUE_UNK:
389	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->sim_pin_value.soft[0];
390	for (unitao = 0; unitao < unit_count; ++unitao) {
391	for (wordao = 0; wordao < 3; ++wordao) {
392	ptrs_ptr->word[wordao] = lm_get_int();
393	}
394	++ptrs_ptr;
395	}
396	instance->restore_state = SENT_RECV_SIM_PIN_VALUE_SOFT;
397	break;
398	case SENT_RECV_SIM_PIN_VALUE_SOFT:
399	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.data[0];
400	for (unitao = 0; unitao < unit_count; ++unitao) {
401	for (wordao = 0; wordao < 3; ++wordao) {
402	ptrs_ptr->word[wordao] = lm_get_int();
403	}
404	++ptrs_ptr;
405	}
406	instance->restore_state = SENT_RECV_SIM_PIN_VALUE_SOFT;
407	break;
408	case SENT_RECV_SIM_PIN_VALUE_SOFT:
409	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz[0];
410	for (unitao = 0; unitao < unit_count; ++unitao) {
411	for (wordao = 0; wordao < 3; ++wordao) {
412	ptrs_ptr->word[wordao] = lm_get_int();
413	}
414	++ptrs_ptr;
415	}
416	instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_DATA;
417	break;
418	case SENT_RECV_LAST_SAMPLE_VALUE_DATA:
419	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.hiz[0];
420	for (unitao = 0; unitao < unit_count; ++unitao) {
421	for (wordao = 0; wordao < 3; ++wordao) {
422	ptrs_ptr->word[wordao] = lm_get_int();
423	}
424	++ptrs_ptr;
425	}
426	instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_HIZ;
427	break;
428	case SENT_RECV_LAST_SAMPLE_VALUE_HIZ:
429	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.unknown[0];
430	for (unitao = 0; unitao < unit_count; ++unitao) {
431	for (wordao = 0; wordao < 3; ++wordao) {
432	ptrs_ptr->word[wordao] = lm_get_int();
433	}
434	++ptrs_ptr;
435	}
436	instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_UNK;
437	break;
438	case SENT_RECV_LAST_SAMPLE_VALUE_UNK:
439	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0];
440	for (unitao = 0; unitao < unit_count; ++unitao) {
441	for (wordao = 0; wordao < 3; ++wordao) {
442	ptrs_ptr->word[wordao] = lm_get_int();
443	}
444	++ptrs_ptr;
445	}
446	instance->restore_state = SENT_RECV_LAST_SAMPLE_VALUE_SOFT;
447	break;
448	case SENT_RECV_LAST_SAMPLE_VALUE_SOFT:
449	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0];
450	for (unitao = 0; unitao < unit_count; ++unitao) {
451	for (wordao = 0; wordao < 3; ++wordao) {
452	ptrs_ptr->word[wordao] = lm_get_int();
453	}
454	++ptrs_ptr;
455	}
456	fix_pel_ctl_word(ptrs_ptr, unitao, dab_ptr);
457	++ptrs_ptr;
458	}
459	write_patterns(instance,
460	instance->unit_addr[instance->cur_unit_addr_index][0],
461	(PTRN_BITS *)instance->last_sample_value.soft[0];
462	if (grow_patterns(instance) == FAILURE)
463	return(FAILURE);
464	instance->restore_state = SENT_RECV_PATTERN;
465	break;
466	case SENT_RECV_PATTERN:
467	ptrs_ptr = (PTRN_BITS_LONGWORD *)instance->last_sample_value.soft[0];
468	for (unitao = 0; unitao < unit_count; ++unitao) {
469	for (wordao = 0; wordao < 3; ++wordao) {
470	ptrs_ptr->word[wordao] = lm_get_int();
471	}
472	++ptrs_ptr;
473	}

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/saverest.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
5/130

LINE #	SOURCE TEXT
481	1
482	1
483	fix_pel_ctl_word(ptra_ptr, unitso, dab_ptr),
484	
485	++ptrs, --;
486	1
487	
488	write_pattern(instance,
489	(instance->unit_addr[instance->cur_unit_addr_index][0],
490	(PTRN_BITS * )temp_ptrn[0]);
491	
492	if (grow_pattern(instance) == FAILURE)
493	return(FAILURE);
494	
495	break;
496	default:
497	break;
498	1
499	
500	return(SUCCESS);
501	1

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/timer.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
1/131

LINE #	SOURCE TEXT
1	/* SCCS ID: timer.c rev 3.1, 4/24/89 at 07:53:56 */
2	
3	#ifndef MODELER
4	#include "sys.h"
5	#include "a/time.h"
6	#include "common.h"
7	
8	static u_long lm_tick = 0;
9	
10	static
11	handle_SIGALRM()
12	{
13	++lm_tick;
14	}
15	
16	static struct itimerval initial_value = { { 0, 20000 }, { 0, 20000 } };
17	
18	inittimer()
19	{
20	struct itimerval oldval;
21	signal(SIGVTALRM, handle_SIGALRM);
22	setitimer(ITIMER_VIRTUAL, &initial_value, &oldval);
23	}
24	
25	lm_time()
26	{
27	return(lm_tick);
28	}
29	
30	lm_delay(delay)
31	{
32	/* "delay" is msec */
33	u_long start;
34	u_long tick;
35	tick = delay / 20 + 1;
36	start = lm_time();
37	while ((lm_time() - start) < tick)
38	;
39	}
40	
41	#endif



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

1/132

```

LINE # SOURCE TEXT
1  /* SCDS_ID: tmeas.c rev 3.4, 5/9/89 at 17:35:09 */
2
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "soprom.h"
7  #include "laserwar.h"
8  #include "protana.h"
9
10 #define MAGIC_OIN_GOUT_DELAY 400
11 #define MAGIC_OIN_IOUT_DELAY 1200
12 #define MAGIC_IIN_GOUT_DELAY -1000
13 #define MAGIC_IIN_IOUT_DELAY -1000
14
15 #define MAX_PIN_NAME_LIST 128
16
17 #ifdef DEBUG
18 #define SWEEP_RANGE 0
19 #else
20 #define SWEEP_RANGE 0
21 #endif
22
23 #define TRY_COUNT 1
24
25 /* The following is used to accumulate all changes from eval/store pins */
26 PTRN_BITS LONGWORD gbl_tm_ident_change[MAX_UNIT_COUNT];
27 PTRN_BITS LONGWORD gbl_tm_ident_inconsistent_pins[MAX_UNIT_COUNT];
28 PTRN_BITS gbl_tm_ident_data_change[MAX_UNIT_COUNT];
29
30 #ifdef DEBUG
31 extern u_char loop_till_key;
32 extern u_long debug_key;
33 extern u_long debug_char;
34 #endif
35
36 /* ABCUSED */
37 tm_evaluate_1_bit_per_pin(uvar, def_ptr, instance,
38                          ident_inconsistent_pins,
39                          steady_state_result,
40                          temp_steady_state_result,
41                          ident_change,
42                          changed_dac)
43
44 USER_INFO *uvar;
45 DEVICE_SPEC *def_ptr;
46 INSTANCE_INFO *instance;
47 PTRN_BITS LONGWORD ident_inconsistent_pins;
48 FULL_VALUE steady_state_result;
49 FULL_VALUE temp_steady_state_result;
50 PTRN_BITS LONGWORD ident_change;
51 u_char changed_dac;
52
53 PIN_INFO *pin_info;
54 EXTRA_DEVICE_SPEC *extra_def_ptr;
55 PTRN_BITS consistent_set[MAX_UNIT_COUNT];
56 PTRN_BITS next_ptrn[MAX_UNIT_COUNT];
57 PTRN_BITS lcyddb[MAX_UNIT_COUNT];
58 PTRN_BITS lcyddb_ptr;
59 PTRN_BITS LONGWORD lcyddb_ptr;
60 PTRN_BITS LONGWORD ident_ios_ptr;
61 PTRN_BITS LONGWORD inst_lcyddb_ptr;
62 PTRN_BITS LONGWORD inst_lcyddb_ptr;
63 DAB_INFO *dab_ptr;
64 PIN_SPEC *pin_def;
65 UMS_OFFSET *ums_ptr;
66 u_long inst_block_number[MAX_LANE_COUNT];
67 u_long next_ptrn_addr[MAX_UNIT_COUNT];
68 u_long last_unit_addr[MAX_LANE_COUNT];
69 u_long seq_end_addr[MAX_LANE_COUNT];
70 u_long timeout;
71 short pin_number;
72 u_short eval_index = 0;
73 u_char pin_value;
74 u_char old_pin_value;
75 u_char allocated_blocks;
76 u_char junk1;
77 u_char junk2;
78 u_char total_unit;
79 u_char unitno;
80 u_char wordno;
81 u_char bitno;
82 u_char evaluation_count = 0;
83 u_char old_val;
84 u_char i;
85 u_char any_driving_to_x;
86 u_char measure;
87
88 DPRINTF(("inside tm_evaluate_1_bit_per_pin\n"));
89
90 /*
91 * The timeout value is set with the assumption that we are running at
92 * 150KHz (1/6.666 usec period).
93 * timeout = pattern play time + 1 second for feedback
94 *          = pattern_count * 8 microsec + 125000 * 8 microsec
95 *          = (pattern_count + 125000) * 8 microsec
96 *          = (pattern_count + 125000) * 8 / 1000 millisecc
97 *          = (pattern_count + 125000) >> 7;
98 */
99
100 timeout = (instance->pattern_count + PTRN_COUNT_FUDGE_FACTOR) >> 7;
101
102 extra_def_ptr = (EXTRA_DEVICE_SPEC *) def_ptr->extra_data;
103 dab_ptr = dab_list(instance->dab_info_index);
104 total_unit = dab_ptr->unit_count;
105
106 clear_ptrn_bits((char *) gbl_tm_ident_change, total_unit);
107 clear_ptrn_bits((char *) gbl_tm_ident_inconsistent_pins, total_unit);
108 clear_ptrn_bits((char *) gbl_tm_ident_data_change, total_unit);
109
110 /*
111 * We don't need to write the LCTCHDB and LCTCHDB patterns here as the
112 * evaluate_1_bit_per_pin() because these will always be written below.
113 */
114
115 init_tm_pin_info_table(gbl_tm_pin_info_table, def_ptr->pin_cnt);
116
117 if (get_next_pattern_addr(instance, next_ptrn_addr,
118                          last_unit_addr, allocated_blocks) == FAILURE)
119     return (FAILURE);
120

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89 PAGE #  
TIME 6:14:50 pm 2/133

```

121 calculate_consistent_set(instance, def_ptr, setup_ptr);
122
123 /* Set drive I/O pins during measurement pattern */
124 lcyddb_ptr = (PTRN_BITS_LONGWORD *) lcyddb;
125 lcyddb_ptr = (PTRN_BITS_LONGWORD *) lcyddb;
126 last_lcyddb_ptr = (PTRN_BITS_LONGWORD *) instance->lcyddb_loaded;
127 last_lcyddb_ptr = (PTRN_BITS_LONGWORD *) instance->lcyddb_loaded;
128 ident_ios_ptr = extra_def_ptr->ident_ios;
129
130 for (unitno = 0; unitno < total_unit; ++unitno) {
131     for (wordno = 0; wordno < 3; ++wordno) {
132
133         lcyddb_ptr->word[wordno] =
134             last_lcyddb_ptr->word[wordno] | ident_ios_ptr->word[wordno];
135
136         lcyddb_ptr->word[wordno] =
137             last_lcyddb_ptr->word[wordno] | ident_ios_ptr->word[wordno];
138     }
139
140     ++lcyddb_ptr;
141     ++lcyddb_ptr;
142     ++last_lcyddb_ptr;
143     ++last_lcyddb_ptr;
144     ++ident_ios_ptr;
145 }
146
147 pin_number = instance->first_data_pin_index;
148 instance->first_data_pin_index = -1;
149
150 while (pin_number != -1) {
151
152     web_ptr = sps_to_short_offset(pin_number);
153     unitno = web_ptr->unitno;
154     wordno = web_ptr->wordno;
155     bitno = web_ptr->bitno;
156
157     set_ptrn_bit(&tbl_tm_ident_data_change(unitno, wordno, bitno));
158
159     pin_def = &def_ptr->pin_table(pin_number);
160     pin_info = &instance->pin_info_table(pin_number);
161     pin_info->input_pin_is_linked = FALSE;
162     pin_value = pin_info->old_filtered;
163
164     /* Keep track of pin changes to build normal pattern sequence. */
165     set_pin_value(instance->pin_pin_value,
166                 unitno, wordno, bitno, pin_value);
167
168     set_measurement_pattern(instance, pin_def,
169                             unitno, wordno, bitno, pin_value,
170                             &junk1, &junk2);
171
172     set_pattern(instance, pin_def,
173                 setup_ptr, unitno, wordno, bitno, pin_value);
174
175     pin_number = pin_info->next_input_pin_index;
176 }
177
178 /* write the setup pattern */
179 write_pattern(instance,
180               &instance->unit_addr[instance->cur_unit_addr_index][0],
181               setup_ptr);
182
183 /* Process the eval pins */
184 for (pin_number = instance->first_eval_pin_index;
185     pin_number != -1;
186     pin_number = pin_info->next_input_pin_index) {
187
188     /* Note that eval pin can only have ENMI format. */
189
190     web_ptr = sps_to_short_offset(pin_number);
191     unitno = web_ptr->unitno;
192     wordno = web_ptr->wordno;
193     bitno = web_ptr->bitno;
194
195     pin_def = &def_ptr->pin_table(pin_number);
196     pin_info = &instance->pin_info_table(pin_number);
197     pin_info->input_pin_is_linked = FALSE;
198     pin_value = pin_info->old_filtered;
199
200     /* Keep track of pin changes to build normal pattern sequence. */
201     set_pin_value(instance->pin_pin_value,
202                 unitno, wordno, bitno, pin_value);
203
204     set_measurement_pattern(instance, pin_def,
205                             unitno, wordno, bitno, pin_value,
206                             &junk1, &junk2);
207
208     old_val = read_ptrn_bit(&setup_ptr[unitno], wordno, bitno);
209
210     set_pattern(instance, pin_def,
211                 setup_ptr, unitno, wordno, bitno, pin_value);
212
213     if (old_val == read_ptrn_bit(&setup_ptr[unitno], wordno, bitno)) {
214
215         /*
216          * If the pin value in the last consistent set is equal to the pin
217          * value in measurement pattern, then we cannot measure this
218          * transition.
219          */
220         DPRINTF(("cannot measure eval pin: %d\n", pin_number));
221         continue;
222     }
223
224     if (eval_index < MAX_EVAL_CHANGES) {
225         gbl_eval_pin_number[eval_index] = (u_short) pin_number;
226         gbl_eval_pin_value[eval_index++] = pin_value;
227     } else {
228         lm_queue_message(WARNING_MSG, "internal error: not enough room to store eval changes; delay number might be incorrect");
229     }
230
231     if (pin_def->direction == IO) {
232         reset_ptrn_bit(&lcyddb[unitno], wordno, bitno);
233         reset_ptrn_bit(&lcyddb[unitno], wordno, bitno);
234     }
235 }
236
237 if (eval_index != 0) {
238     /* evaluation complete */
239     write_pattern(instance, &instance->lcyddb_addr[0], lcyddb);
240     write_pattern(instance, &instance->lcyddb_addr[0], lcyddb);

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE	5/23/89	PAGE #
TIME	6:14:50 pm	3/134

```

LINE #          SOURCE TEXT
241      next_off_output(instance, setup_ptr);
242      write_pattern(instance, next_ptr_addr, setup_ptr);
243
244
245      set_seq_end_bit(instance,
246                      last_unit_addr,
247                      allocated_blocks,
248                      seq_end_addr,
249                      inst_block_number);
250
251      if (measure_delay(def_ptr, instance, timeout,
252                      EVAL_EVENT, gbl_eval_pin_number, eval_index,
253                      steady_state_result,
254                      ident_inconsistent_pins,
255                      temp_steady_state_result,
256                      gbl_pin_info_table,
257                      ident_change, changed_dac) == FAILURE) {
258          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
259
260          if (allocated_blocks == TRUE)
261              return_last_blocks(instance);
262
263          return (FAILURE);
264      }
265      remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
266
267      /* Soft drive the IO EVAL pins for next evaluation */
268      for (i = 0; i < eval_index; ++i) {
269          pin_number = gbl_eval_pin_number[i];
270          if (def_ptr->pin_table[pin_number].direction == IO) {
271              uwb_ptr = uwb_ptr + short_offset[pin_number];
272              unitno = uwb_ptr->unitno;
273              wordno = uwb_ptr->wordno;
274              bitno = uwb_ptr->bitno;
275
276              set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
277              set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
278          }
279      }
280
281      /* Restore the sequence to normal state. */
282      write_pattern(instance,
283                  instance->lcychdb_addr[0],
284                  instance->lcychdb_loaded);
285      write_pattern(instance,
286                  instance->lcychdb_addr[0],
287                  instance->lcychdb_loaded);
288      write_pattern(instance,
289                  instance->unit_addr[instance->cur_unit_addr_index][0],
290                  instance->ptrn_loaded);
291
292      /*
293       * Run normal pattern history and check that the result obtained is the
294       * same as with timing measurement pattern history.
295       */
296
297      set_seq_end_bit(instance,
298                      instance->last_addr,
299                      FALSE,
300                      seq_end_addr,
301                      inst_block_number);
302
303      if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
304          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
305
306          if (allocated_blocks == TRUE)
307              return_last_blocks(instance);
308
309          return (FAILURE);
310      }
311
312      if (get_result(def_ptr, instance, ident_change,
313                  EVAL_EVENT, gbl_eval_pin_number,
314                  eval_index, last_driving_to_1) == FAILURE) {
315          remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
316
317          if (allocated_blocks == TRUE)
318              return_last_blocks(instance);
319
320          return (FAILURE);
321      }
322
323      check_consistency2(instance,
324                      instance->last_sample_value,
325                      steady_state_result,
326                      ident_inconsistent_pins,
327                      ident_change,
328                      total_unit);
329
330      for (i = 0; i < eval_index; ++i) {
331          pin_number = gbl_eval_pin_number[i];
332          if (def_ptr->pin_table[pin_number].direction == IO) {
333              uwb_ptr = uwb_ptr + short_offset[pin_number];
334              unitno = uwb_ptr->unitno;
335              wordno = uwb_ptr->wordno;
336              bitno = uwb_ptr->bitno;
337
338              reset_ptrn_bit(&ident_inconsistent_pins[unitno], wordno, bitno);
339
340              /*
341               * For timing measurement the eval pins will be sampled as driving
342               * (because we turn on LCYCHDB) and sampled as 1 for regular
343               * pattern history, and therefore the ident_change will be set for
344               * these eval pins. These are bogus changes so we need to reset the
345               * ident_change bit for these pins.
346               */
347              reset_ptrn_bit(&ident_change[unitno], wordno, bitno);
348          }
349      }
350
351      for (i = 0; i < eval_index; ++i) {
352          /*
353           * Get the table delays for all pins. Some of the delays will be
354           * overridden later if it turned out that the pin delays can be
355           * measured. This get_delay() loop has to be after we reset the
356           * ident_change bit for the eval pins because otherwise we would see
357           * unspec delays from one eval pin to another eval pin.
358           */
359      }
360

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE	5/23/89	PAGE #
TIME	6:14:50 pm	4/135

```

LINE # SOURCE TEXT
361 get_delay(def_ptr, instance, ident_change,
362 ident_inconsistent_pin,
363 gbl_eval_pin_number[i], gbl_eval_pin_value[i]);
364 }
365
366 /* Accumulate the pin changes in gbl_tm_ident_change
367 accumulate_ident_pins(ident_change, gbl_tm_ident_change, total_unit);
368
369 copy_ptrn_bits((char *) ident_inconsistent_pins,
370 (char *) gbl_tm_ident_inconsistent_pins,
371 total_unit);
372
373 add_inconsistent_pins(instance, ident_inconsistent_pins);
374
375 clear_ptrn_bits((char *) ident_change, total_unit);
376 clear_ptrn_bits((char *) ident_inconsistent_pins, total_unit);
377
378 if (any_driving_to_x == TRUE) {
379
380 /*
381 * Use 2 bits per pin --> hard drive those JO STORE pins which change
382 * from driving to X.
383 */
384
385 if (def_ptr->device_type == PUBLIC) {
386
387 /*
388 * Write BUDGENA_L0ADDR to previous pattern address. For PRIVATE
389 * devices we don't have to do anything since BUDGENA will always be
390 * written when we do the next evaluation.
391 */
392
393 write_pattern(instance,
394 instance->unit_addr[
395 instance->cur_unit_addr_index + 1 & 1][0],
396 instance->busmb_loaded);
397
398 write_pattern(instance,
399 instance->unit_addr[instance->cur_unit_addr_index][0],
400 instance->last_consistent_set);
401
402 if (grow_pattern(instance) == FAILURE)
403 return (FAILURE);
404 }
405
406 remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
407
408 if (allocated_blocks == TRUE)
409 return_last_blocks(instance);
410
411 /* Process the STORE pins */
412 for (pin_number = instance->first_store_pin_index,
413 instance->first_store_pin_index = -1,
414 pin_number != -1,
415 pin_number = pin_info->next_input_pin_index) {
416
417 gbl_eval_pin_number[0] = (u_short) pin_number;
418
419 uwb_ptr = &pe_to_short_offset[pin_number];
420 unitno = uwb_ptr->unitno;
421 wordno = uwb_ptr->wordno;
422 bitno = uwb_ptr->bitno;
423
424 pin_info = instance->pin_info_table[pin_number];
425 pin_def = def_ptr->pin_table[pin_number];
426
427 pin_info->input_pin_is_linked = FALSE;
428 pin_value = pin_info->old_filtered;
429
430 if (pin_def->direction == IN) {
431 /* INPUT STORE change */
432
433 old_pin_value = read_pin_value(instance->pin_pin_value,
434 unitno, wordno, bitno);
435
436 set_pin_value(instance->pin_pin_value,
437 unitno, wordno, bitno, pin_value);
438
439 switch (pin_def->clk_format) {
440 case RNI:
441 case INRZ:
442
443 if (input_pin_transition(old_pin_value, pin_value,
444 pin_info->uninitialized_pin) == NO_TRANSITION) {
445
446 /*
447 * This is an error because the host should have filtered the
448 * transition. On second thought it is NOT an error because
449 * the host only filters transitions to exactly the same
450 * value.
451 */
452
453 DPRINTF(("No transition on IN STORE INRZ pin %s",
454 pin_def->pin_name));
455 continue;
456
457 if (pin_value & (LOGIC_0 | LOGIC_50 | LOGIC_10))
458 reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
459 else
460 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
461 break;
462
463 case RI:
464
465 if (input_pin_transition(old_pin_value, pin_value,
466 pin_info->uninitialized_pin) != RISE_TRANSITION) {
467 DPRINTF(("No transition on IN STORE RI pin %s",
468 pin_def->pin_name));
469 continue;
470 }
471
472 /* Disable the RI clock on the measurement pattern (ptrn_loaded) */
473 set_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
474 break;
475
476 case RO:
477
478 if (input_pin_transition(old_pin_value, pin_value,
479 pin_info->uninitialized_pin) != FALL_TRANSITION) {
480 DPRINTF(("No transition on IN STORE RO pin %s",
481 pin_def->pin_name));
482 continue;

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

5/136

```

481
482 /* Disable the R1 clock on the measurement pattern (ptrn_loaded) */
483 reset_ptrn_bit(instance->ptrn_loaded[unitno], wordno, bitno);
484
485 break;
486 default:
487     in_queue_message(ERROR_MSG, "internal error: illegal pin type in device.b");
488     continue;
489 }
490 } else {
491     /* IO STORE change */
492     /* Note: IO store pin can only have DMSZ format */
493     old_pin_value = read_pin_value(instance->last_sample_value,
494                                     unitno, wordno, bitno);
495     set_pin_value(instance->pin_value,
496                  unitno, wordno, bitno, pin_value);
497
498     if (!io_pin_transition(old_pin_value, pin_value,
499                          pin_info->uninitialized_pin == NO_TRANSITION)) {
500         DPRINTF(("no transition on IO STORE pin %s", pin_def->pin_name));
501         continue;
502     }
503     set_measurement_pattern(instance, pin_def,
504                             unitno, wordno, bitno, pin_value,
505                             &junk1, &junk2);
506 }
507
508 if (instance->word_2_bit_per_pin) {
509     switch_to_2_bit_per_pin(instance);
510 }
511
512 measure = TRUE;
513 switch (pin_def->clk_format) {
514     case R1:
515         write_pattern(instance,
516                      instance->unit_addr(instance->cur_unit_addr_index)[0],
517                      setup_ptrn);
518         if (grow_pattern(instance) == FAILURE)
519             return (FAILURE);
520         reset_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
521         break;
522     case R0:
523         write_pattern(instance,
524                      instance->unit_addr(instance->cur_unit_addr_index)[0],
525                      setup_ptrn);
526         if (grow_pattern(instance) == FAILURE)
527             return (FAILURE);
528         set_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
529         break;
530     default:
531         old_val = read_ptrn_bit(setup_ptrn[unitno], wordno, bitno);
532         write_pattern(instance,
533                      instance->unit_addr(instance->cur_unit_addr_index)[0],
534                      setup_ptrn);
535         set_pattern(instance, pin_def,
536                    setup_ptrn, unitno, wordno, bitno, pin_value);
537         if (old_val == read_ptrn_bit(setup_ptrn[unitno], wordno, bitno)) {
538             DPRINTF(("cannot measure store PN: %d\n", pin_number));
539             measure = FALSE;
540             if (grow_pattern(instance) == FAILURE)
541                 return (FAILURE);
542             break;
543         }
544     }
545
546     if (pin_def->direction == IO) {
547         /* Always hard and medium drive the worst pin */
548         reset_ptrn_bit(&lcychdb[unitno], wordno, bitno);
549         reset_ptrn_bit(&lcychdb[unitno], wordno, bitno);
550     }
551     if (measure == TRUE)
552         ++evaluation_count;
553     write_pattern(instance, instance->lcychdb_addr[0], lcychdb);
554     write_pattern(instance, instance->lcychdb_addr[0], lcychdb);
555     mask_off_output(instance, setup_ptrn);
556     write_pattern(instance,
557                  instance->unit_addr(instance->cur_unit_addr_index)[0],
558                  setup_ptrn);
559     if (measure == TRUE) {
560         set_seq_end_bit(instance,
561                        instance->seq_end_addr,
562                        FALSE,
563                        seq_end_addr,
564                        inst_block_number);
565     }
566     if (measure_delay(def_ptr, instance, timeout,
567                     STORE_EVENT, (u_short *) &pin_number, 1,
568                     steady_state_result,
569                     ident_inconsistent_pins,
570                     temp_steady_state_result,
571                     gbl_pin_info_table,
572                     ident_change, changed_dac) == FAILURE) {
573         remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
574         return (FAILURE);
575     }
576     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
577 }
578 /* Sort drive this IO pin for next evaluation */
579 if (pin_def->direction == IO) {
580     set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
581     set_ptrn_bit(&lcychdb[unitno], wordno, bitno);
582 }
583 /* Make the pattern history look like normal pattern history */
584 calculate_consistent_set(instance, def_ptr, consistent_set);
585 switch (pin_def->clk_format) {

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89 PAGE #  
TIME 6:14:50 pm 6/137

```

LINE # SOURCE TEXT
601 case R1:
602     reset_ptrn_bit(&consistent_set[unitno], wordno, bitno);
603     /* Disable the R1 clock on the setup ptrn so it will be
604      * correct when we evaluation the next store change in
605      * this simulation pass.
606     */
607     set_ptrn_bit(&setup_ptrn[unitno], wordno, bitno);
608     break;
609 case R0:
610     set_ptrn_bit(&consistent_set[unitno], wordno, bitno);
611     /* Disable the R0 clock on the setup ptrn so it will be
612      * correct when we evaluation the next store change in
613      * this simulation pass.
614     */
615     reset_ptrn_bit(&setup_ptrn[unitno], wordno, bitno);
616     break;
617 default:
618     break;
619 }
620
621 /* save the consistent set */
622 copy_ptrn_bits((char *) consistent_set,
623               (char *) instance->last_consistent_set,
624               ddb_list(instance)->ddb_info_index->unit_count);
625
626 write_pattern(instance, &instance->lcycmbb_addr[0],
627               instance->lcycmbb_loaded);
628
629 write_pattern(instance, &instance->lcycmbb_addr[0],
630               instance->lcycmbb_loaded);
631
632 write_pattern(instance,
633               &instance->
634               unit_addr[instance->cur_unit_addr_index + 1 & 1][0],
635               consistent_set);
636
637 write_pattern(instance,
638               &instance->unit_addr[instance->cur_unit_addr_index][0],
639               instance->ptrn_loaded);
640
641 set_seq_end_bit(instance,
642               instance->seq_end_addr,
643               FALSE,
644               seq_end_addr,
645               inst_block_number);
646
647
648 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
649     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
650     return (FAILURE);
651 }
652 if (get_result(def_ptr, instance, ident_change,
653               STORE_EVENT, gbl_eval_pin_number,
654               1, &any_driving_to_x) == FAILURE) {
655     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
656     return (FAILURE);
657 }
658 if (measure == TRUE) {
659     check_consistency(instance,
660                       instance->last_sample_value,
661                       steady_state_result,
662                       ident_inconsistent_pins,
663                       ident_change,
664                       total_unit);
665 }
666 reset_ptrn_bit(&ident_inconsistent_pins[unitno], wordno, bitno);
667 reset_ptrn_bit(&ident_change[unitno], wordno, bitno);
668
669 get_delay(def_ptr, instance, ident_change,
670           ident_inconsistent_pins,
671           (u_short) pin_number, pin_value);
672
673 /* Accumulate the pin changes in gbl_tm_ident_change */
674 accumulate_ident_pins(ident_change, gbl_tm_ident_change, total_unit);
675
676 copy_ptrn_bits((char *) ident_inconsistent_pins,
677               (char *) gbl_tm_ident_inconsistent_pins,
678               total_unit);
679
680 add_inconsistent_pins(instance, ident_inconsistent_pins);
681
682 clear_ptrn_bits((char *) ident_change, total_unit);
683 clear_ptrn_bits((char *) ident_inconsistent_pins, total_unit);
684
685 if (any_driving_to_x == TRUE) {
686     /*
687      * Use 2 bits per pin --> hard drive those IO STORE pins which change
688      * from driving to 3.
689     */
690
691     if (def_ptr->device_type == PUBLIC) {
692         /*
693          * Write HMODEM_LOADED to previous pattern address. For PRIVATE
694          * devices we don't have to do anything since HMODEM will always be
695          * written when we do the next evaluation.
696         */
697         write_pattern(instance,
698                       &instance->unit_addr[
699                       instance->cur_unit_addr_index + 1 & 1][0],
700                       instance->hmodeb_loaded);
701
702         write_pattern(instance,
703                       &instance->unit_addr[instance->cur_unit_addr_index][0],
704                       instance->last_consistent_set);
705
706         if (grow_pattern(instance) == FAILURE)
707             return (FAILURE);
708     }
709     remove_seq_end_bit(instance, seq_end_addr, inst_block_number);
710
711     pin_info->uninitialized_pin = FALSE;
712 }
713
714 if (evaluation_count == 0)
715     return (SUCCESS);
716
717 copy_tm_result(instance, gbl_tm_pin_info_table,

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
7/138

```

LINE #          SOURCE TEXT
721          gbl_tm_ident_change, gbl_tm_ident_inconsistent_pins);
722
723      return (SUCCESS);
724  }
725
726  set_pattern(instance, pin_def, ptrn_ptr, unitno, wordno, bitno, pin_value)
727  INSTANCE_INFO *instance;
728  PIN_SPEC      *pin_def;
729  PTRN_BITS     *ptrn_ptr;
730  u_char        unitno;
731  u_char        wordno;
732  u_char        bitno;
733  u_char        pin_value;
734  {
735      u_char      last_sample_value;
736
737      if (pin_def->direction == IN) {
738          if (pin_value & (LOGIC_0 | LOGIC_50))
739              reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
740          else
741              set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
742      } else {
743          /* IO pin -> let the DUT pin dominate. */
744          last_sample_value =
745              read_pin_value(instance->last_sample_value, unitno, wordno, bitno);
746
747          if (last_sample_value & LOGIC_0)
748              reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
749          else if (last_sample_value & LOGIC_1)
750              set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
751          else if (last_sample_value & (LOGIC_10 | LOGIC_21)) {
752              if (pin_value & (LOGIC_0 | LOGIC_50))
753                  reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
754              else if (pin_value & (LOGIC_1 | LOGIC_51))
755                  set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
756              else if (last_sample_value & LOGIC_10)
757                  reset_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
758              else
759                  set_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
760          }
761          else {
762              /* Last sample is U */
763              toggle_ptrn_bit(ptrn_ptr[unitno], wordno, bitno);
764          }
765      }
766      last_sample_value =
767          read_pin_value(instance->last_sample_value, unitno, wordno, bitno);
768  }
769
770  copy_tm_result(instance, tm_pin_info_table, ident_change,
771                ident_inconsistent_pins)
772  INSTANCE_INFO *instance;
773  TM_PIN_INFO   *tm_pin_info_table;
774  PTRN_BITS_LONGWORD *ident_change;
775  PTRN_BITS_LONGWORD *ident_inconsistent_pins;
776  {
777      DAB_INFO      *dab_ptr;
778      TM_PIN_INFO   *tm_pin_info;
779      PIN_INFO      *pin_info;
780      u_long        mask;
781      u_long        ident_change_word;
782      u_short        unit_pin_number_offset;
783      u_short        word_pin_number_offset;
784      u_short        dest_pin_number;
785      u_char         dest_pin_value;
786      u_char         total_unit;
787      u_char         unitno;
788      u_char         wordno;
789      u_char         bitno;
790
791      dab_ptr = dab_list(instance->dab_info_index);
792      total_unit = dab_ptr->unit_count;
793      unit_pin_number_offset = 0;
794      for (unitno = 0; unitno < total_unit; ++unitno) {
795          word_pin_number_offset = unit_pin_number_offset + 79;
796
797          for (wordno = 0; wordno < 3; ++wordno) {
798              ident_change_word = ident_change[unitno].word[wordno];
799
800              for (bitno = 31; bitno >= 0; --bitno) {
801                  if (ident_change_word == 0)
802                      break;
803                  mask = bitno_to_mask(bitno);
804                  if (ident_change_word & mask) {
805                      /* Reset the bit */
806                      ident_change_word ^= mask;
807                      dest_pin_number = word_pin_number_offset + bitno - 31;
808                      tm_pin_info = tm_pin_info_table[dest_pin_number];
809                      pin_info = instance->pin_info_table[dest_pin_number];
810                      if (read_ptrn_bit_long(ident_inconsistent_pins[unitno],
811                                           wordno, (u_char) bitno) != 0) {
812                          dest_pin_value = LOGIC_U;
813                          tm_pin_info->delay_value_is_set = FALSE;
814                      } else {
815                          dest_pin_value =
816                              read_pin_value(instance->last_sample_value,
817                                              unitno, wordno, (u_char) bitno);
818                      }
819
820                      pin_info->new_val = dest_pin_value;
821                      if (tm_pin_info->delay_value_is_set == TRUE) {
822                          /* Return measured delay */
823                          DPRINTF(("measured PN: %d val: %d delay: %d-%d\n",
824                                  dest_pin_number,
825                                  dest_pin_value,
826                                  tm_pin_info->min_delay,
827                                  tm_pin_info->max_delay));
828
829                          if (pin_info->output_pin_is_linked == FALSE) {
830                              pin_info->next_output_pin_index =
831                                  instance->first_data_pin_index;
832                              instance->first_data_pin_index = dest_pin_number;
833                          }
834                      }
835                  }
836              }
837          }
838      }
839  }
840

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/tmeas.c	DATE 5/23/89	PAGE # 8/139
LINE #		SOURCE TEXT		
841		pin_info->output_pin_is_linked = TRUE;		
842		pin_info->event_pin_number =		
843		tm_pin_info->event_pin_number;		
844		pin_info->delay_type = MEASURED;		
845				
846		pin_info->min_delay = tm_pin_info->min_delay;		
847		pin_info->max_delay = tm_pin_info->max_delay;		
848		} else {		
849		/* pin delay was not measured */		
850		PRINTF(("Not measured PM: id val: %d\n",		
851		dest_pin_number,		
852		dest_pin_value));		
853		}		
854		}		
855		}		
856		word_pin_number_offset -- 32;		
857		}		
858				
859		unit_pin_number_offset -- 80;		
860		}		
861		}		
862				
863		get_next_pattern_addr(instance, new_unit_addr,		
864		last_unit_addr, allocated_blocks)		
865		INSTANCE_INFO		
866		/*instance,		
867		/*new_unit_addr,		
868		/*last_unit_addr,		
869		/*allocated_blocks,		
870		{		
871		/* ??? */		
872		DAB_INFO		
873		/*lane_info,		
874		/*new_block_addr[MAX_LANE_COUNT],		
875		/*cur_unit_addr,		
876		/*addr_inc_per_lane,		
877		/*laneno,		
878		/*unitno,		
879		/*total_unit,		
880		/*junk.		
881		dab_ptr = dab_list(instance->dab_info_index);		
882		addr_inc_per_lane = dab_ptr->unit_count_per_lane * PTRN_ADDR_INC;		
883		total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;		
884				
885		/*allocated_blocks = FALSE;		
886		for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno)		
887		new_block_addr[laneno] = 0;		
888				
889		cur_unit_addr = (instance->unit_addr(instance->cur_unit_addr_index)[0];		
890		for (unitno = 0; unitno < total_unit; ++unitno) {		
891		laneno = dab_ptr->unit_location[unitno].lane_no;		
892		lane_info = (instance->lane_addr[laneno];		
893		if ((cur_unit_addr[unitno] + addr_inc_per_lane) >=		
894		lane_info->max_addr) {		
895		if (new_block_addr[laneno] == 0) {		
896		if (new_block_addr[laneno] == 0) {		
897		/*junk,		
898		/*PATTERN_BLOCK,		
899		/*laneno) == FAILURE) {		
900		in_queue_message(ERROR MSG, "out of Fast Pattern Memory");		
901		instance->failed_to_alloc_ptrn = TRUE;		
902		return (FAILURE);		
903				
904		/*allocated_blocks = TRUE;		
905		set_branch(lane_info->max_addr - PTRN_ADDR_INC,		
906		new_block_addr[laneno]);		
907				
908		new_unit_addr[unitno] = new_block_addr[laneno] +		
909		cur_unit_addr[unitno] + addr_inc_per_lane -		
910		lane_info->max_addr;		
911		} else {		
912		new_unit_addr[unitno] = cur_unit_addr[unitno] + addr_inc_per_lane;		
913				
914				
915		if (dab_ptr->unit_location[unitno].last_in_lane)		
916		last_unit_addr[laneno].last_unit_addr = new_unit_addr[unitno];		
917				
918				
919		return (SUCCESS);		
920		}		
921				
922		return last_blocks(instance);		
923		INSTANCE_INFO		
924		{		
925				
926		/*		
927		/* Release the blocks allocated after the current unit addr.		
928		/* instance->lane_addr contains information related to the real pattern, not		
929		/* the temporary pattern. We find the block to be released by looking at the		
930		/* link table.		
931		*/		
932				
933		DAB_INFO		
934		/*lane_info,		
935		/*blocknum_to_be_released,		
936		/*laneno,		
937		{		
938		dab_ptr = dab_list(instance->dab_info_index);		
939				
940		for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) {		
941		lane_info = (instance->lane_addr[laneno];		
942				
943		if (dab_ptr->lane_used[laneno]) {		
944		blocknum_to_be_released =		
945		read_branch_table_content(lane_info->max_addr - PTRN_ADDR_INC);		
946				
947		release_block((u_long) (LANE_0_START_ADDR + laneno * LANE_ADDR_INC +		
948		(blocknum_to_be_released << BLOCK_NUMBER_SHIFT), laneno);		
949				
950		unset_branch(lane_info->max_addr - PTRN_ADDR_INC);		
951				
952		}		
953				
954				
955		measure_delay(def_ptr, instance, timeout,		
956		event_type, event_pin_number, event_pin_count,		
957		steady_state_result, ident_inconsistent_pins,		
958		temp_steady_state_result, tm_pin_info_table,		
959		ident_change, changed_dec)		
960		DEVICE_SPEC		
		/*def_ptr,		



Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

9/140

```

LINE # SOURCE TEXT
961 INSTANCE_INFO *instance;
962 u_long timeout;
963 u_char event_type;
964 u_short event_pin_number;
965 u_char event_pin_count;
966 FULL_VALUE *steady_state_result;
967 PTRN_BITS LONGWORD *ident_inconsistent_pins;
968 FULL_VALUE *temp_steady_state_result;
969 TM_PIN_INFO *tm_pin_info_table;
970 PTRN_BITS LONGWORD *ident_change;
971 u_char *changed_dac;
972 {
973
974
975 /* Measure delay of all output pins which change as a result of the pin
976 * event_pin_number(s). The type of the event (EVAL_EVENT or STORE_EVENT) is
977 * given in "event_type". If event_type == EVAL_EVENT then there could be
978 * multiple eval pins that change on this particular pattern seq. We need to
979 * add the size of the eval pins to the measured pin delay that we found.
980 */
981 PTRN_BITS LONGWORD ident_all_pin_change[MAX_UNIT_COUNT];
982 PTRN_BITS LONGWORD *ident_all_pin_change_ptr;
983 PTRN_BITS LONGWORD *last_sample_data_ptr;
984 PTRN_BITS LONGWORD *last_sample_hiz_ptr;
985 PTRN_BITS LONGWORD *last_sample_unknown_ptr;
986 PTRN_BITS LONGWORD *steady_state_data_ptr;
987 PTRN_BITS LONGWORD *steady_state_hiz_ptr;
988 PTRN_BITS LONGWORD *steady_state_unknown_ptr;
989 PTRN_BITS LONGWORD *ident_ioe_ptr;
990 PTRN_BITS LONGWORD *ident_outputs_ptr;
991 PTRN_BITS LONGWORD *two_state_result[MAX_UNIT_COUNT];
992 DAB_INFO *dab_ptr;
993 TM_PIN_INFO *tm_pin_info;
994 TM_PIN_INFO *tm_pin_info_ptr;
995 EXTRA_DEVICE_SPEC *extra_def_ptr;
996 DMS_OFFSET *dms_ptr;
997 u_long ident_change_word;
998 u_long mask;
999 long min_event_pin_delay;
1000 long max_event_pin_delay;
1001 long min_delay;
1002 long max_delay;
1003 long save_min_delay;
1004 long save_max_delay;
1005 u_long z_to_1_transition;
1006 u_long measurement_error;
1007 short pins_to_be_measured_head;
1008 short all_pins_to_be_measured_head = -1;
1009 short pin_number;
1010 u_short min_event_pin_number;
1011 u_char resolution;
1012 u_char total_pins_to_be_measured = 0;
1013 u_char i;
1014 u_char total_unit;
1015 u_char waitno;
1016 u_char wordno;
1017 char hitno;
1018 u_char measurement_mode;
1019
1020 /* COMMENTED CODE
1021 char pin_name_list[MAX_PIN_NAME_LIST];
1022 u_char valid_pin_name_list = FALSE;
1023 END COMMENTED CODE
1024
1025 DPRINTF(("inside measure_delay\n"));
1026
1027 extra_def_ptr = (EXTRA_DEVICE_SPEC *) def_ptr->extra_data;
1028 dab_ptr = dab_list(instance->dab_info_index);
1029 total_unit = dab_ptr->unit_count;
1030
1031 #ifdef DBASE
1032 setup_final_tm_result(total_unit * 80);
1033 #endif
1034
1035 is_measure_delay = TRUE;
1036
1037 if (event_type == EVAL_EVENT) {
1038 /* Program the sample edge to start at data time */
1039 measurement_mode = extra_def_ptr->eval_event_mode;
1040 extra_def_ptr->edge_setting[6] = extra_def_ptr->edge_setting[8];
1041 if (is_tm2_set_sample_trigger_mode(
1042 (u_long)extra_def_ptr->eval_event_mode) == FAILURE) {
1043 in_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
1044 return (FAILURE);
1045 }
1046 } else {
1047 /* Program the sample edge to start at store time */
1048 measurement_mode = extra_def_ptr->store_event_mode;
1049 extra_def_ptr->edge_setting[6] = extra_def_ptr->edge_setting[7];
1050 if (is_tm2_set_sample_trigger_mode(
1051 (u_long)extra_def_ptr->store_event_mode) == FAILURE) {
1052 in_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
1053 return (FAILURE);
1054 }
1055 }
1056
1057 /* Sample the outputs at the max range of the finest resolution */
1058 if (set_edge_and_sample_setting(extra_def_ptr,
1059 (u_long) MAX_SAMPLE_RANGE - SWEEP_RANGE) == FAILURE)
1060 return (FAILURE);
1061
1062 if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE)
1063 return (FAILURE);
1064
1065 DPRINTF(("get end range result, resolution: %d\n", RESOLUTION_05_NS));
1066
1067 read_magic_full_sample_reg(instance, steady_state_result);
1068 read_magic_timing_sample_reg(instance, two_state_result,
1069 MAX_SAMPLE_RANGE - SWEEP_RANGE);
1070
1071 ident_all_pin_change_ptr = ident_all_pin_change;
1072 last_sample_data_ptr = (PTRN_BITS LONGWORD *)
1073 instance->last_sample_value.data;
1074 last_sample_hiz_ptr = (PTRN_BITS LONGWORD *)
1075 instance->last_sample_value.hiz;
1076 last_sample_unknown_ptr = (PTRN_BITS LONGWORD *)
1077 instance->last_sample_value.unknown;
1078
1079
1080

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

lm1000/tmeas.c

DATE 5/23/89

PAGE #

TIME 6:14:50 pm

10/141

```

1081 steady_state_data_ptr = (PTM_BITS_LONGWORD *)
1082 steady_state_result_data;
1083 steady_state_hiz_ptr = (PTM_BITS_LONGWORD *)
1084 steady_state_result_hiz;
1085 steady_state_unknown_ptr = (PTM_BITS_LONGWORD *)
1086 steady_state_result_unknown;
1087
1088 ident_ios_ptr = (PTM_BITS_LONGWORD *) extra_def_ptr->ident_ios;
1089 ident_outputs_ptr = (PTM_BITS_LONGWORD *) extra_def_ptr->ident_outputs;
1090
1091 for (unitno = 0; unitno < total_unit; ++unitno) {
1092     for (wordno = 0; wordno < 3; ++wordno) {
1093         ident_all_pin_change_ptr->word[wordno] =
1094             last_sample_data_ptr->word[wordno];
1095         steady_state_data_ptr->word[wordno];
1096
1097         /* Don't measure transitions on I/O pins */
1098         ident_all_pin_change_ptr->word[wordno] |=
1099             (last_sample_hiz_ptr->word[wordno] |
1100              last_sample_unknown_ptr->word[wordno] |
1101              steady_state_hiz_ptr->word[wordno] |
1102              steady_state_unknown_ptr->word[wordno]);
1103
1104         /* Measure ZL to H transition on I/O pins */
1105         z_to_h_transition =
1106             ident_ios_ptr->word[wordno] &
1107             (last_sample_hiz_ptr->word[wordno] &
1108              last_sample_data_ptr->word[wordno]) &
1109             (~steady_state_hiz_ptr->word[wordno] |
1110              ~steady_state_unknown_ptr->word[wordno]);
1111         ident_all_pin_change_ptr->word[wordno] |= z_to_h_transition;
1112
1113         if (z_to_h_transition) {
1114             DPRINTF(("Found ZL to H transition\n"));
1115         }
1116         /* Measure H to ZL transition on I/O pins */
1117         h_to_zl_transition =
1118             ident_ios_ptr->word[wordno] &
1119             (last_sample_hiz_ptr->word[wordno] &
1120              last_sample_data_ptr->word[wordno]) &
1121             (~steady_state_hiz_ptr->word[wordno] |
1122              ~steady_state_unknown_ptr->word[wordno]);
1123         ident_all_pin_change_ptr->word[wordno] |= h_to_zl_transition;
1124
1125         if (h_to_zl_transition) {
1126             DPRINTF(("Found H to ZL transition\n"));
1127         }
1128         /* Only measure timing on I/O pins and outputs pins */
1129         ident_all_pin_change_ptr->word[wordno] |=
1130             ident_ios_ptr->word[wordno] | ident_outputs_ptr->word[wordno];
1131
1132         ++ident_all_pin_change_ptr;
1133         ++last_sample_data_ptr;
1134         ++last_sample_hiz_ptr;
1135         ++last_sample_unknown_ptr;
1136         ++steady_state_data_ptr;
1137         ++steady_state_hiz_ptr;
1138         ++steady_state_unknown_ptr;
1139         ++ident_ios_ptr;
1140         ++ident_outputs_ptr;
1141     }
1142 }
1143
1144 /* Reset the bits in ident_all_pin_change corresponding to the
1145  * event pin number. These pins will be forced added later. This is
1146  * applicable only for IO pins because input pins have been masked off.
1147  */
1148 for (i = 0; i < event_pin_count; ++i) {
1149     pin_number = event_pin_number[i];
1150
1151     word_ptr = &word_ptr->wordno;
1152     wordno = word_ptr->wordno;
1153     bitno = word_ptr->bitno;
1154
1155     reset_ptr->bit((PTM_BITS *) ident_all_pin_change)[unitno],
1156         wordno, (u_char) bitno);
1157 }
1158
1159 /* Find all pins which change values and link them in tm_pin_info_table */
1160 for (unitno = 0; unitno < total_unit; ++unitno) {
1161     for (wordno = 0; wordno < 3; ++wordno) {
1162         ident_change_word = ident_all_pin_change[unitno].word[wordno];
1163         for (bitno = 31; bitno >= 0; --bitno) {
1164             if (ident_change_word & (1 << bitno)) {
1165                 break;
1166             }
1167
1168             mask = bitno_to_mask(bitno);
1169
1170             if (ident_change_word & mask) {
1171                 ident_change_word ^= mask;
1172
1173                 --total_pins_to_be_measured;
1174
1175                 pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
1176
1177                 DPRINTF(("link to tm_pin_info_table PW: %d\n", pin_number));
1178
1179                 tm_pin_info = &tm_pin_info_table[pin_number];
1180                 tm_pin_info->delay_found = FALSE;
1181
1182                 tm_pin_info->next_pin_index = all_pins_to_be_measured_head;
1183                 all_pins_to_be_measured_head = pin_number;
1184             }
1185         }
1186     }
1187 }
1188
1189 /* Always measure the delay of the event pins */
1190 for (i = 0; i < event_pin_count; ++i) {
1191     event_pin = &tm_pin_info_table[event_pin_number[i]];
1192 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE	5/23/89	PAGE #
TIME	6:14:50 pm	11/142

```

LINE #          SOURCE TEXT
1201      event_pin->delay_found = FALSE;
1202      DPRINTF(("link eval pin to tm_pin_info_table PN: %d\n",
1203              event_pin_number[i]));
1204
1205      /* Use the minimum pin number to identify the event pin */
1206      if (event_pin_number[i] < min_event_pin_number)
1207          min_event_pin_number = event_pin_number[i];
1208
1209      event_pin->next_pin_index = all_pins_to_be_measured_head;
1210      all_pins_to_be_measured_head = event_pin_number[i];
1211  }
1212
1213      resolution = RESOLUTION_QS_MS;
1214
1215      while (resolution != RESOLUTION_INVALID) {
1216
1217          /*
1218           * The actual pin delays are referenced to the event pin delay on the
1219           * same sample ramp. So we need to measure the event pins for each
1220           * resolution.
1221           */
1222
1223          /* Add the event pins to the list of pins to be measured */
1224          for (i = 0; i < event_pin_count; ++i) {
1225              event_pin = &tm_pin_info_table[event_pin_number[i]];
1226              event_pin->delay_found = FALSE;
1227              ++total_pins_to_be_measured;
1228          }
1229
1230      find_pins_to_be_measured(def_ptr,
1231                              tm_pin_info_table, two_state_result,
1232                              steady_state_result,
1233                              ident_all_pins_change,
1234                              event_pin_number,
1235                              event_pin_count,
1236                              spins_to_be_measured_head,
1237                              total_wait, resolution, measurement_mode);
1238
1239      pin_number = pins_to_be_measured_head;
1240      while (pin_number != -1) {
1241
1242          uwb_ptr = &pin_to_short_offset[pin_number];
1243          unitno = uwb_ptr->unitno;
1244          wordno = uwb_ptr->wordno;
1245          bitno = uwb_ptr->bitno;
1246
1247          tm_pin_info_table[pin_number].event_pin_number =
1248              min_event_pin_number;
1249
1250          if (read_ptrn_bit(&(PTRN_BITS *) ident_inconsistent_pins)(unitno),
1251              wordno, (u_char) bitno) == 0) {
1252              if (measure_pin(instance, timeout,
1253                              tm_pin_info_table, (u_short) pin_number,
1254                              steady_state_result, ident_inconsistent_pins,
1255                              temp_steady_state_result, ident_change,
1256                              resolution, changed_dac, measurement_mode) == FAILURE)
1257                  return (FAILURE);
1258          } else {
1259              DPRINTF(("PN: %d inconsistent --> not measured\n",
1260                      pin_number));
1261          }
1262
1263          --total_pins_to_be_measured;
1264          pin_number = tm_pin_info_table[pin_number].next_pin_to_be_measured;
1265      }
1266
1267      if (get_composite_eval_pin_delay(def_ptr, instance, tm_pin_info_table,
1268                                      event_pin_number, event_pin_count,
1269                                      measurement_mode, resolution,
1270                                      min_event_pin_delay, max_event_pin_delay) == FAILURE)
1271          return (FAILURE);
1272
1273      pin_number = pins_to_be_measured_head;
1274      while (pin_number != -1) {
1275          tm_pin_info = &tm_pin_info_table[pin_number];
1276
1277          /* Make believe that the delay is found, so that we will be
1278           * able to report delay not reached correctly.
1279           * We will look at the actual delay values to see whether
1280           * we can/can't measure the pin delay.
1281           */
1282          tm_pin_info->delay_found = TRUE;
1283
1284          lm_tmg_get_sample_ramp_delay(resolution,
1285                                      tm_pin_info->min_threshold,
1286                                      min_delay,
1287                                      measurement_error);
1288
1289          min_delay -= measurement_error;
1290
1291          lm_tmg_get_sample_ramp_delay(resolution,
1292                                      tm_pin_info->max_threshold,
1293                                      max_delay,
1294                                      measurement_error);
1295          max_delay += measurement_error;
1296
1297          if ((min_delay < max_event_pin_delay) ||
1298              (max_delay < min_event_pin_delay)) {
1299              /*-----*/
1300              /* This is bogus delay (we failed to measure the pin transition).
1301               * This case happens when:
1302               * 1. We try to measure a transition from
1303               *    0 to 1 (or 1 to 0) on a DATA pin, but the simulator is
1304               *    sending a 1 (or 0) with the clock transition which causes
1305               *    the pin to switch from 0 to 1 (or 1 to 0). The transition
1306               *    from 0 to 1 (or 1 to 0) will already happen in the setup
1307               *    patterns, and therefore we will NOT be able to measure the
1308               *    transition.
1309               * 2. There is a misdeclaration of a STORE or EVAL pin
1310               *    in the Shell Software as shown below:
1311               *    DATA (a/b store/eval) _____
1312               *    STORE (causing event) _____
1313               *    OUT _____
1314               *    We fail to measure transition on OUT because it happens
1315               *    in the setup patterns.
1316               * 3. A pin is inconsistent between two sets of sample runs.
1317               *    Consider this scenario. Store pin A change to 1 and we
1318               *    sample output B to be 1, then store pin A change to 0
1319               *    and we sample output B to be 0. At this point we say
1320               *    that we should measure the transition on pin B.
1321

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89 PAGE #  
TIME 6:14:50 pm 12/143

```

1321      /* It's possible that we will fail to measure B because
1322      * this time pin B does not change to 1 on the rising
1323      * edge of pin A.
1324      * .....
1325      */
1326      /* COERCED CODE: case 1 above
1327      for (i = 0; i < event_pin_count; ++i) {
1328          if (pin_number == event_pin_number[i])
1329              break;
1330      }
1331
1332      if (i == event_pin_count) {
1333          pin_number = tm_pin_info->next_pin_to_be_measured;
1334          continue;
1335      }
1336
1337      tm_ptr = tm_ptr->short_offset(pin_number);
1338      ushort = tm_ptr->ushort;
1339      ushort = tm_ptr->ushort;
1340      bitno = tm_ptr->bitno;
1341
1342      if (read_pin_bit((PWR_BITS *)ident_inconsistent_pins)(ushort),
1343          ushort, (u_char)bitno) != 0) {
1344          pin_number = tm_pin_info->next_pin_to_be_measured;
1345          continue;
1346      }
1347
1348      if (read_pin_bit((PWR_BITS *)ident_data_change)(ushort),
1349          ushort, bitno) == 0) {
1350
1351          if (build_pin_name_list == FALSE) {
1352              build_pin_list(def_ptr, total_unit,
1353                  (PWR_BITS_LONGWORD *)gbl_tm_ident_data_change,
1354                  pin_name_list);
1355              build_pin_name_list = TRUE;
1356          }
1357
1358          in_yname_namemap(KBERR_YNS, "could not measure pin: is; one of these pins should be declared as store/eval; is",
1359              def_ptr->pin_table(pin_number).pin_name,
1360              pin_name_list);
1361      }
1362      /* COERCED CODE */
1363      } else {
1364
1365          save_min_delay = tm_pin_info->min_delay;
1366          save_max_delay = tm_pin_info->max_delay;
1367
1368          /* The real delay is referenced to the skew of the eval pins */
1369          tm_pin_info->min_delay = min_delay - max_event_pin_delay;
1370          tm_pin_info->max_delay = max_delay - min_event_pin_delay;
1371
1372          /* Adjust the delay further by the delay of the trace */
1373          adjust_measured_delay(def_ptr->pin_table, tm_pin_info, (u_short) pin_number, tm_pin_info_table);
1374
1375          if (tm_pin_info->delay_value_is_set == FALSE) {
1376              tm_pin_info->delay_value_is_set = TRUE;
1377          } else {
1378              /* Coalesce delays from several runs of timing measurements. */
1379              if (save_min_delay < tm_pin_info->min_delay)
1380                  tm_pin_info->min_delay = save_min_delay;
1381              if (save_max_delay > tm_pin_info->max_delay)
1382                  tm_pin_info->max_delay = save_max_delay;
1383          }
1384      }
1385
1386      pin_number = tm_pin_info->next_pin_to_be_measured;
1387  }
1388
1389  if (total_pins_to_be_measured == 0) {
1390      break;
1391  }
1392  /* Go to the next coarser resolution */
1393  switch (resolution) {
1394      case RESOLUTION_05_NS:
1395          resolution = RESOLUTION_10_NS;
1396          break;
1397      case RESOLUTION_10_NS:
1398          resolution = RESOLUTION_20_NS;
1399          break;
1400      case RESOLUTION_20_NS:
1401          resolution = RESOLUTION_40_NS;
1402          break;
1403      case RESOLUTION_40_NS:
1404          resolution = RESOLUTION_INVALID;
1405          break;
1406      default:
1407          resolution = RESOLUTION_INVALID;
1408          continue;
1409  }
1410
1411  if (set_edge_and_sample_setting(extra_def_ptr,
1412      (u_long) resolution,
1413      (u_long) MAX_SAMPLE_RANGE - SWEEP_RANGE) == FAILURE)
1414      return (FAILURE);
1415
1416  DPRINTF(("get end range result: resolution: %d\n", resolution));
1417
1418  if (play_ptr_seq(instance, timeout, changed_dac) == FAILURE)
1419      return (FAILURE);
1420
1421  read_magic_full_sample_res(instance, temp_steady_state_result);
1422  read_magic_timing_sample_res(instance, two_state_result,
1423      MAX_SAMPLE_RANGE - SWEEP_RANGE);
1424
1425  /* Note that "ident_change" below will be overwritten by get_result()
1426  * which we are going to call later on. This is OK since we only
1427  * collecting the "ident_inconsistent_pins". The above note does not apply
1428  * anymore since we are collecting the ident_change in get_result().
1429  */
1430  check_consistency(instance,
1431      steady_state_result, temp_steady_state_result,
1432      ident_inconsistent_pins, ident_change,
1433      total_unit);
1434  }
1435
1436  /* Set the delays for event pins to 0 */
1437  for (i = 0; i < event_pin_count; ++i) {
1438      event_pin = tm_pin_info_table[event_pin_number[i]];
1439      event_pin->delay_value_is_set = FALSE;
1440      event_pin->min_delay = 0;
  
```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
13/144

```

1441 event_pin->max_delay = 0;
1442 }
1443
1444 /*
1445 * Check that all pins are measured. Note that "event_pin" is pointing to
1446 * the last event pin.
1447 */
1448 pin_number = event_pin->next_pin_index;
1449 while (pin_number != -1) {
1450     tm_pin_info = tm_pin_info_table(pin_number);
1451     if (tm_pin_info->delay_found == FALSE) {
1452         uwb_ptr = tps_to_short_offset(pin_number);
1453         unitno = uwb_ptr->unitno;
1454         wordno = uwb_ptr->wordno;
1455         bitno = uwb_ptr->bitno;
1456
1457         if (read_ptr_bit(&(PTRN_BITS *) ident_inconsistent_pins)(unitno),
1458             wordno, (u_char) bitno) == 0)
1459             in_queue_message(WARNING_MSG, "pin: %s of instance: %s delay not reached",
1460                             def_ptr->pin_table(pin_number).pin_name,
1461                             instance->device_info_string);
1462     }
1463     pin_number = tm_pin_info->next_pin_index;
1464 }
1465
1466 is_measure_delay = FALSE;
1467 DPRINTF(("Exiting measure_delay\n"));
1468 return (SUCCESS);
1469 }
1470
1471 find_pins_to_be_measured(def_ptr,
1472                          tm_pin_info_table, end_range_two_state_result,
1473                          steady_state_result, ident_all_pin_change,
1474                          event_pin_number, event_pin_count,
1475                          pins_to_be_measured_head,
1476                          total_unit, resolution, measurement_mode)
1477
1478 DEVICE_SPEC *def_ptr;
1479 TM_PIN_INFO *tm_pin_info_table;
1480 PTRN_BITS_LONGWORD *steady_state_result;
1481 FULL_VALUE *ident_all_pin_change;
1482 PTRN_BITS_LONGWORD *event_pin_number;
1483 u_short *event_pin_count;
1484 u_char *pins_to_be_measured_head;
1485 short *total_unit;
1486 u_char *resolution;
1487 u_char *measurement_mode;
1488
1489 {
1490     PTRN_BITS_LONGWORD *steady_state_data_ptr;
1491     PTRN_BITS_LONGWORD *ident_change_ptr;
1492     PTRN_BITS_LONGWORD *ident_change[MAX_UNIT_COUNT];
1493     TM_PIN_INFO *tm_pin_info;
1494     UWB_OFFSET *uwb_ptr;
1495     u_long *ident_change_word;
1496     u_long *mask;
1497     u_short *pin_number;
1498     u_char *min_range;
1499     u_char *min_sample_ramp0;
1500     u_char *min_sample_ramp1;
1501     u_char *min_sample_ramp2;
1502     u_char *min_sample_ramp3;
1503     u_char *i;
1504     u_char *unitno;
1505     u_char *wordno;
1506     char *bitno;
1507
1508     DPRINTF(("inside find_pins_to_be_measured\n"));
1509
1510     in_tm_get_minimum_sample_ramp_setting(&min_sample_ramp0,
1511                                           &min_sample_ramp1,
1512                                           &min_sample_ramp2,
1513                                           &min_sample_ramp3);
1514
1515     switch (resolution) {
1516         case RESOLUTION_05_MS:
1517             min_range = min_sample_ramp0 + SKEEP_RANGE;
1518             break;
1519         case RESOLUTION_10_MS:
1520             min_range = min_sample_ramp1 + SKEEP_RANGE;
1521             break;
1522         case RESOLUTION_20_MS:
1523             min_range = min_sample_ramp2 + SKEEP_RANGE;
1524             break;
1525         case RESOLUTION_40_MS:
1526             min_range = min_sample_ramp3 + SKEEP_RANGE;
1527             break;
1528         default:
1529             min_range = min_sample_ramp3 + SKEEP_RANGE;
1530             break;
1531     }
1532
1533     steady_state_data_ptr = (PTRN_BITS_LONGWORD *) steady_state_result->data;
1534     ident_change_ptr = (PTRN_BITS_LONGWORD *) ident_change;
1535
1536     /*
1537     * Identify the pins that have attained their steady state value at the end
1538     * of the sample ramp.
1539     */
1540     for (unitno = 0; unitno < total_unit; ++unitno) {
1541         for (wordno = 0; wordno < 3; ++wordno) {
1542             ident_change_ptr[word[wordno]] =
1543                 (steady_state_data_ptr[word[wordno]] -
1544                  end_range_two_state_result[word[wordno]]) &
1545                 ident_all_pin_change->word[wordno];
1546         }
1547         ++steady_state_data_ptr;
1548         ++ident_change_ptr;
1549         ++end_range_two_state_result;
1550         ++ident_all_pin_change;
1551     }
1552
1553     steady_state_data_ptr = (PTRN_BITS_LONGWORD *) steady_state_result->data;
1554     pins_to_be_measured_head = -1;
1555     for (unitno = 0; unitno < total_unit; ++unitno) {
1556         for (wordno = 0; wordno < 3; ++wordno) {
1557             ident_change_word = ident_change[unitno].word[wordno];
1558             for (bitno = 31; bitno >= 0; --bitno) {
1559                 if (ident_change_word == 0)
1560                     break;
1561                 mask = bitno to mask[bitno];
1562             }
1563         }
1564     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
14/145

```

1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680

SOURCE TEXT

if (idest_change_word & mask) {
    idest_change_word = mask;

    pin_number = CALC_PIN_NUMBER_LONG(unitno, wordno, bitno);
    DPRINTF(("measure this pin in cur res PW: %d\n",
        pin_number));

    tm_pin_info = tm_pin_info_table(pin_number);

    if (tm_pin_info->delay_found == FALSE) {
        tm_pin_info->expected_value =
            read_ptr_bit_long(&steady_state_data_ptr(unitno),
                wordno, (u_char) bitno);
        tm_pin_info->min_threshold = MIN_RANGE;
        tm_pin_info->max_threshold = MAX_SAMPLE_RANGE -
            SWEEP_RANGE;

        tm_pin_info->next_pin_to_be_measured =
            "pin to be measured head";
        "pin to be measured head" = pin_number;
    } else {
        DPRINTF(("PW: %d delay is already found; not measured\n",
            pin_number));
    }
}

/* Link the event pins to the next_pin_to_be_measured. */
for (i = 0; i < event_pin_count; ++i) {
    pin_number = event_pin_number[i];

    uwb_ptr = &uwb_ptr_offset(pin_number);
    unitno = uwb_ptr->unitno;
    wordno = uwb_ptr->wordno;
    bitno = uwb_ptr->bitno;

    DPRINTF(("measure this pin in cur res PW: %d\n",
        pin_number));

    tm_pin_info = tm_pin_info_table(pin_number);
    tm_pin_info->expected_value =
        read_ptr_bit(&steady_state_result->data(unitno),
            wordno, (u_char) bitno);

    if ((measurement_mode == EARLYSAMPLETRIGGERMODE) &&
        ((def_ptr->pin_table(pin_number).clk_format == R1) ||
        (def_ptr->pin_table(pin_number).clk_format == R0))) {
        tm_pin_info->min_threshold =
            ((EXTRA_DEVICE_SPEC *) def_ptr->extra_data->ri_or_r2_min_threshold;
    } else {
        tm_pin_info->min_threshold = min_range;
    }

    tm_pin_info->max_threshold = MAX_SAMPLE_RANGE - SWEEP_RANGE;

    tm_pin_info->next_pin_to_be_measured =
        "pin to be measured head";
    "pin to be measured head" = pin_number;
}

DPRINTF(("exiting find_pins_to_be_measured\n"));

}

/*
NAME:
    round_trip_trace_delay -- Returns the round trip delay for this DAB.
PURPOSE:
    "round_trip_trace_delay" calculates the delay attributed to the propagation
    of the signal between the ERT and the magic chip. This includes the delay
    from the magic chip to the event pin, the delay from the result pin back to
    the magic chip and the PEL delay for driven pins.
INTERFACE:
    u_short round_trip_trace_delay(pin_def, event_pin, result_pin)
PARAMETERS      TYPE      DESCRIPTION
pin_def         u_short    A pointer to the pin specification table.
event_pin       u_short    The pin number of the event pin.
result_pin      u_short    The pin number of the result pin.
CALLS:
    None.
EXTERNAL REFERENCES:
    None.
RESTRICTIONS:
    No checking is done to validate any of the parameters.
DESIGNER:
    Steve Parks
*/

#define PEL_DELAY 400
u_short
round_trip_trace_delay(pin_def, event_pin, result_pin)
register PIN_SPEC *pin_def,
u_short event_pin,
u_short result_pin;

/*
Returns composite trace delay for this adapter (in Ps).
*/
{
    return (PEL_DELAY
        + pin_def[event_pin].trace_delay
        + pin_def[result_pin].trace_delay);
}

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE	5/23/89	PAGE #
TIME	6:14:50 pm	15/146

```

1681 *
1682 * NAME:
1683 *   adjust_measured_delay -- Compute actual delay from measured delay.
1684 *
1685 * PURPOSE:
1686 *   The "adjust_measured_delay" adjusts the delay values for the
1687 *   maximum and minimum delays for "other" known delays not attributed
1688 *   to actual DUT. It also adjusts the delays attributed to the
1689 *   MAGIC IC delay effect.
1690 *   In the event that the measured delay is less than
1691 *   the "other" known delays, the measured delay will be set to zero.
1692 *
1693 * INTERFACE:
1694 *
1695 *   adjust_measured_delay(pin_def, tm_pin_info,
1696 *       result_pin, tm_pin_info_table)
1697 *
1698 *   PARAMETER      TYPE      DESCRIPTION
1699 *   -----
1700 *   pin_def         PIN_SPEC*  A pointer to the pin specification table.
1701 *   tm_pin_info     TM_PIN_INFO* A pointer to the timing measurement pin
1702 *       information table.
1703 *   result_pin      u_short     The pin number of the result pin.
1704 *   tm_pin_info_table TM_PIN_INFO* A pointer to the TM_PIN_INFO table
1705 *       to find the event pin's TM_PIN_INFO.
1706 *
1707 * CALLS:
1708 *   round_trip_trace_delay()
1709 *
1710 * EXTERNAL REFERENCES:
1711 *   None.
1712 *
1713 * RESTRICTIONS:
1714 *   No checking is done to validate any of the parameters.
1715 *
1716 * DESIGNER:
1717 *   Steve Parke
1718 *
1719 * MODIFICATIONS:
1720 *   05/29/89: RRM Added Magic delay adjustments
1721 *
1722 * .....
```

```

1723 *
1724 * adjust_measured_delay(pin_def, tm_pin_info, result_pin, tm_pin_info_table)
1725 * register PIN_SPEC *pin_def;
1726 * register TM_PIN_INFO *tm_pin_info;
1727 * u_short result_pin;
1728 * TM_PIN_INFO *tm_pin_info_table;
1729 * {
1730 * {
1731 *
1732 * /*
1733 * * The delay we measured is actually longer than it should be because of
1734 * * trace delays on the PDL and the DAB.
1735 * */
1736 * u_short event_pin = tm_pin_info->event_pin_number;
1737 * u_short trace_delay = round_trip_trace_delay(pin_def, event_pin, result_pin);
1738 * TM_PIN_INFO *event_tm_pin_info;
1739 * short magic_delay;
1740 *
1741 * DPRINTF(("Delay (%5s,%5s): ", pin_def[event_pin].pin_name, pin_def[result_pin].pin_name));
1742 * DPRINTF(("Before: %5d,%5d: ", tm_pin_info->min_delay, tm_pin_info->max_delay));
1743 *
1744 * tm_pin_info->min_delay = (trace_delay > tm_pin_info->min_delay)
1745 *     ? 0 : tm_pin_info->min_delay - trace_delay;
1746 * tm_pin_info->max_delay = (trace_delay > tm_pin_info->max_delay)
1747 *     ? 0 : tm_pin_info->max_delay - trace_delay;
1748 *
1749 * DPRINTF(("After: %5d,%5d: ", tm_pin_info->min_delay, tm_pin_info->max_delay));
1750 *
1751 * /* Adjust the delay further by the MAGIC IC effect. */
1752 * event_tm_pin_info = tm_pin_info_table[event_pin];
1753 * if (event_tm_pin_info->expected_value) {
1754 *     if (tm_pin_info->expected_value) {
1755 *         magic_delay = MAGIC_IIN_LOUT_DELAY;
1756 *     }
1757 *     else {
1758 *         magic_delay = MAGIC_IIN_OOUT_DELAY;
1759 *     }
1760 * }
1761 * else {
1762 *     if (tm_pin_info->expected_value) {
1763 *         magic_delay = MAGIC_OIN_LOUT_DELAY;
1764 *     }
1765 *     else {
1766 *         magic_delay = MAGIC_OIN_OOUT_DELAY;
1767 *     }
1768 * }
1769 *
1770 * tm_pin_info->min_delay =
1771 *     ((long)tm_pin_info->min_delay + (long)magic_delay < 0)
1772 *     ? 0 : tm_pin_info->min_delay + magic_delay;
1773 * tm_pin_info->max_delay =
1774 *     ((long)tm_pin_info->max_delay + (long)magic_delay < 0)
1775 *     ? 0 : tm_pin_info->max_delay + magic_delay;
1776 *
1777 * DPRINTF(("After2: %5d,%5d:", tm_pin_info->min_delay, tm_pin_info->max_delay));
1778 * }
1779 *
1780 * measure_pin(instance, timeout, tm_pin_info_table, target_pin_number,
1781 *     steady_state_result, ident_inconsistent_pins,
1782 *     temp_steady_state_result, ident_change, resolution,
1783 *     changed_dac, measurement_mode)
1784 * {
1785 *     u_long instance;
1786 *     TM_PIN_INFO *tm_pin_info;
1787 *     u_short target_pin_number;
1788 *     FULL_VALUE *steady_state_result;
1789 *     PTRN_BITS_LONGWORD *ident_inconsistent_pins;
1790 *     FULL_VALUE *temp_steady_state_result;
1791 *     PTRN_BITS_LONGWORD *ident_change;
1792 *     u_char resolution;
1793 *     u_char changed_dac;
1794 *     u_char measurement_mode;
1795 * {
1796 *     DEVICE_SPEC *dev_ptr;
1797 *     EXTRA_DEVICE_SPEC *extra_dev_ptr;
1798 *     DAB_INFO *dab_ptr;
1799 *     TM_PIN_INFO *tm_pin_info;
1800 *     TM_PIN_INFO *target_pin;
```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/tmeas.c	DATE 5/23/89	PAGE # 16/147
LINE #		SOURCE TEXT		
1801	UWB_OFFSET	*uwb_ptr,		
1802	PTM_BITS	two_state_result[MAX_UNIT_COUNT];		
1803	u_long	temp;		
1804	short	pin_number;		
1805	short	start;		
1806	short	end;		
1807	short	mid;		
1808	u_char	total_unit;		
1809	u_char	result;		
1810	u_char	unitno;		
1811	u_char	wordno;		
1812	u_char	bitno;		
1813	u_char	min_sample_ramp0;		
1814	u_char	min_sample_ramp1;		
1815	u_char	min_sample_ramp2;		
1816	u_char	min_sample_ramp3;		
1817	u_char	min_range;		
1818				
1819		DPRINTF(("inside measure_pin PW: %d\n", target_pin_number));		
1820				
1821		dab_ptr = dab_list(instance->dab_info_index);		
1822		total_unit = dab_ptr->unit_count;		
1823				
1824		def_ptr = instance->definition;		
1825		extra_def_ptr = (EXTRA_DEVICE_SPEC *) instance->definition->extra_data;		
1826				
1827		target_pin = stm_pin_info_table[target_pin_number];		
1828				
1829		start = target_pin->min_threshold;		
1830		end = target_pin->max_threshold;		
1831		mid = start + (end - start) / 2;		
1832				
1833		while (start <= end) {		
1834		/* Sample the pins at "mid" */		
1835				
1836		if (set_edge_and_sample_setting(extra_def_ptr,		
1837		(u_long) resolution,		
1838		(u_long) mid) == FAILURE)		
1839		return (FAILURE);		
1840				
1841		DPRINTF(("getting two state sample; sample time: %d\n", mid));		
1842				
1843		if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE)		
1844		return (FAILURE);		
1845				
1846		#ifdef DEBUG		
1847		if (loop_till_key == TRUE) {		
1848		printf("looping for threshold: %d\n", mid);		
1849		debug_key = 0;		
1850		while (debug_key == 0) {		
1851		if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE)		
1852		return (FAILURE);		
1853				
1854		if (debug_key == (u_long) 't') {		
1855		loop_till_key = FALSE;		
1856		printf("stop loop\n");		
1857				
1858		debug_key = 0;		
1859				
1860		#endif		
1861				
1862		read_magic_full_sample_req(instance, temp_steady_state_result);		
1863		read_magic_timing_sample_req(instance, two_state_result, (u_short) mid);		
1864				
1865		/*		
1866		* Note that "ident_change" below will be overwritten by get_result();		
1867		* which we are going to call later on. This is OK since we only		
1868		* collect the "ident_inconsistent_pins". The above note does not apply		
1869		* anymore since we are collecting the ident_change in get_result().		
1870		*/		
1871		check_consistency(instance,		
1872		steady_state_result, temp_steady_state_result,		
1873		ident_inconsistent_pins, ident_change,		
1874		total_unit);		
1875				
1876				
1877		/* Narrow the range for all pins to be measured except the target pin */		
1878		pin_number = target_pin->next_pin_to_be_measured;		
1879		while (pin_number != -1) {		
1880				
1881		uwb_ptr = aps_to_short_offset(pin_number);		
1882		unitno = uwb_ptr->unitno;		
1883		wordno = uwb_ptr->wordno;		
1884		bitno = uwb_ptr->bitno;		
1885				
1886		tm_pin_info = stm_pin_info_table[pin_number];		
1887				
1888		result = read_ptrn_bit(two_state_result[unitno], wordno, bitno);		
1889				
1890		if (result == tm_pin_info->expected_value) {		
1891		if (mid < tm_pin_info->max_threshold)		
1892		tm_pin_info->max_threshold = mid;		
1893		} else {		
1894		if (mid > tm_pin_info->min_threshold)		
1895		tm_pin_info->min_threshold = mid;		
1896		}		
1897				
1898		pin_number = tm_pin_info->next_pin_to_be_measured;		
1899				
1900				
1901		uwb_ptr = aps_to_short_offset(target_pin_number);		
1902		unitno = uwb_ptr->unitno;		
1903		wordno = uwb_ptr->wordno;		
1904		bitno = uwb_ptr->bitno;		
1905				
1906		/* Adjust the range of the target pin */		
1907		result = read_ptrn_bit(two_state_result[unitno], wordno, bitno);		
1908		if (result == target_pin->expected_value) {		
1909		target_pin->max_threshold = mid;		
1910		end = mid - 1;		
1911		} else {		
1912		target_pin->min_threshold = mid;		
1913		start = mid + 1;		
1914		}		
1915				
1916		mid = start + (end - start) / 2;		
1917				
1918				
1919		if (target_pin->min_threshold > target_pin->max_threshold) {		
1920		/* Exchange the delay */		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
17/148

```

1621      temp = target_pin->min_threshold;
1622      target_pin->min_threshold = target_pin->max_threshold;
1623      target_pin->max_threshold = temp;
1624  }
1625  DPRINTF(("delay found: %d - %d\n",
1626          target_pin->min_threshold, target_pin->max_threshold));
1627
1628  if ((measurement_mode == EARLYSAMPLETRIGGERMODE) &&
1629      ((def_ptr->pin_table[target_pin_number].clk_format == R1) ||
1630       (def_ptr->pin_table[target_pin_number].clk_format == R0))) {
1631      min_range = entire_def_ptr->ri_or_rz_min_threshold;
1632  } else {
1633      lm_tmg_get_minimum_sample_ramp_setting(&min_sample_ramp0,
1634                                             &min_sample_ramp1,
1635                                             &min_sample_ramp2,
1636                                             &min_sample_ramp3);
1637
1638      switch (resolution) {
1639      case RESOLUTION_05_NS:
1640          min_range = min_sample_ramp0;
1641          break;
1642      case RESOLUTION_10_NS:
1643          min_range = min_sample_ramp1;
1644          break;
1645      case RESOLUTION_20_NS:
1646          min_range = min_sample_ramp2;
1647          break;
1648      case RESOLUTION_40_NS:
1649          min_range = min_sample_ramp3;
1650          break;
1651      default:
1652          min_range = 0xfff;
1653          break;
1654      }
1655  }
1656
1657  uwb_ptr = tpa_to_short_offset(target_pin_number);
1658  unitno = uwb_ptr->unitno;
1659  wordno = uwb_ptr->wordno;
1660  bitno = uwb_ptr->bitno;
1661
1662  result = read_ptr_bits(&two_state_result[unitno], wordno, bitno);
1663  if (result == target_pin->expected_value) {
1664      if (target_pin->min_threshold <= min_range + SWEEP_RANGE) {
1665          DPRINTF(("pin: %d --> delay not found\n", target_pin_number));
1666          target_pin->delay_found = FALSE;
1667      } else {
1668          target_pin->delay_found = TRUE;
1669      }
1670  } else {
1671      target_pin->delay_found = TRUE;
1672  }
1673
1674  DPRINTF(("delay found after sweep: %d - %d\n",
1675          target_pin->min_threshold, target_pin->max_threshold));
1676
1677  DPRINTF(("exiting measure_pin\n"));
1678  return (SUCCESS);
1679  }
1680
1681  #ifdef DBASE
1682  setup_final_tm_result(total_pin_count)
1683  u_short      total_pin_count;
1684  {
1685      u_short      i;
1686      long         pin_number;
1687      u_long       expected_time;
1688      char         line[80];
1689
1690      DPRINTF(("enter final tm result, end with pin number -- 1\n"));
1691
1692      for (i = 0; i < total_pin_count; ++i) {
1693          tm_result_array[i].resolution = RESOLUTION_INVALID;
1694          tm_result_array[i].expected_time = MAX_SAMPLE_RAMP_RANGE - SWEEP_RANGE;
1695          tm_result_array[i].set_by_user = FALSE;
1696      }
1697
1698      while (1) {
1699          DPRINTF(("pin number: "));
1700          gets(line);
1701          sscanf(line, "%d", &pin_number);
1702
1703          if (pin_number == -1)
1704              break;
1705
1706          DPRINTF(("expected value: "));
1707          gets(line);
1708
1709          if (strcmp(line, "0") == 0) {
1710              tm_result_array[pin_number].steady_state_full_value = LOGIC_0;
1711              tm_result_array[pin_number].two_state_value = LOGIC_0;
1712          } else if (strcmp(line, "1") == 0) {
1713              tm_result_array[pin_number].steady_state_full_value = LOGIC_1;
1714              tm_result_array[pin_number].two_state_value = LOGIC_1;
1715          } else if (strcmp(line, "10") == 0) {
1716              tm_result_array[pin_number].steady_state_full_value = LOGIC_10;
1717              tm_result_array[pin_number].two_state_value = LOGIC_0;
1718          } else if (strcmp(line, "11") == 0) {
1719              tm_result_array[pin_number].steady_state_full_value = LOGIC_11;
1720              tm_result_array[pin_number].two_state_value = LOGIC_1;
1721          } else if (strcmp(line, "01") == 0) {
1722              tm_result_array[pin_number].steady_state_full_value = LOGIC_01;
1723              tm_result_array[pin_number].two_state_value = LOGIC_0;
1724          } else if (strcmp(line, "00") == 0) {
1725              tm_result_array[pin_number].steady_state_full_value = LOGIC_00;
1726              tm_result_array[pin_number].two_state_value = LOGIC_0;
1727          } else {
1728              DPRINTF(("illegal value. ignored\n"));
1729              continue;
1730          }
1731
1732          DPRINTF(("expected time: "));
1733          gets(line);
1734          sscanf(line, "%d", &expected_time);
1735
1736          tm_result_array[pin_number].resolution = RESOLUTION_05_NS;
1737          tm_result_array[pin_number].expected_time = expected_time;
1738          tm_result_array[pin_number].set_by_user = TRUE;
1739      }
1740  }
1741  #endif

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/tmeas.c	DATE 5/23/89	PAGE # 18/149
LINE #		SOURCE TEXT		
2041		accumulate_idest pins(source, dest, total_unit)		
2042		PTRN_BITS LONGWORD *source;		
2043		PTRN_BITS LONGWORD *dest;		
2044		u_char total_unit;		
2045		{		
2046		u_long *source_ptr;		
2047		u_long *dest_ptr;		
2048		u_char total_word;		
2049		u_char wordao;		
2050		{		
2051		total_word = total_unit * 3;		
2052		source_ptr = (u_long *) source;		
2053		dest_ptr = (u_long *) dest;		
2054		for (wordao = 0; wordao < total_word; ++wordao)		
2055		{		
2056		dest_ptr[wordao]  = source_ptr[wordao];		
2057		}		
2058		/* COMMENTED CODE		
2059		build_pin_list(def_ptr, total_unit, idest_data_change_ptr, pin_name_list)		
2060		DEVICE_SPEC *def_ptr;		
2061		u_char total_unit;		
2062		PTRN_BITS LONGWORD *idest_data_change_ptr;		
2063		char *pin_name_list;		
2064		{		
2065		{		
2066		u_long mask;		
2067		u_long idest_data_change;		
2068		u_short unit_pin_number_offset;		
2069		u_short word_pin_number_offset;		
2070		u_short pin_number;		
2071		u_char unitao;		
2072		u_char wordao;		
2073		u_char bitao;		
2074		u_char max_string_length;		
2075		{		
2076		max_string_length = MAX_PIN_NAME_LIST - 5;		
2077		{		
2078		pin_name_list[0] = '\0';		
2079		unit_pin_number_offset = 0;		
2080		for (unitao = 0; unitao < total_unit; ++unitao) {		
2081		{		
2082		word_pin_number_offset = unit_pin_number_offset + 79;		
2083		for (wordao = 0; wordao < 3; ++wordao) {		
2084		{		
2085		idest_data_change = idest_data_change_ptr[unitao].word[wordao];		
2086		for (bitao = 31; bitao >= 0; --bitao) {		
2087		{		
2088		mask = bitao < 31 ? 1 : 0;		
2089		if (idest_data_change & mask) {		
2090		{		
2091		pin_number = word_pin_number_offset + bitao + 31;		
2092		{		
2093		if (strlen(pin_name_list) +		
2094		strlen(def_ptr->pin_table[pin_number].pin_name) + 1 <=		
2095		max_string_length) {		
2096		{		
2097		def_ptr->pin_table[pin_number].pin_name,		
2098		{		
2099		{		
2100		{		
2101		{		
2102		{		
2103		{		
2104		{		
2105		{		
2106		{		
2107		{		
2108		{		
2109		{		
2110		{		
2111		{		
2112		{		
2113		{		
2114		END COMMENTED CODE		
2115		/*		
2116		{		
2117		get_composite_eval_pin_delay(def_ptr, instance, tm_pin_info_table,		
2118		event_pin_number, event_pin_count,		
2119		measurement_mode, resolution,		
2120		comp_eval_min_delay, comp_eval_max_delay)		
2121		{		
2122		DEVICE_SPEC *def_ptr;		
2123		INSTANCE *instance;		
2124		TM_PIN_INFO *tm_pin_info_table;		
2125		u_short event_pin_number;		
2126		u_char event_pin_count;		
2127		u_char measurement_mode;		
2128		u_char resolution;		
2129		long comp_eval_min_delay;		
2130		long comp_eval_max_delay;		
2131		{		
2132		u_char i;		
2133		TM_PIN_INFO *event_pin;		
2134		long min_event_pin_delay;		
2135		long max_event_pin_delay;		
2136		u_long measurement_error;		
2137		{		
2138		/* Convert the range found in measure_pin() to real delay in picosec */		
2139		/* Find the min-max skew of all EVAL pins in terms of register setting */		
2140		min_event_pin_delay = MAX_SAMPLE_RANGE - SKEW_RANGE;		
2141		max_event_pin_delay = 0;		
2142		{		
2143		if (measurement_mode == EDGE_TRIGGERMODE) {		
2144		{		
2145		event_pin = tm_pin_info_table[event_pin_number[i]];		
2146		{		
2147		if (event_pin->delay_found == FALSE) {		
2148		{		
2149		in_queue_message(ERROR_MSG, "internal error: event pin: %s of instance: %s delay not found",		
2150		instance->device_info_string);		
2151		return (FAILURE);		
2152		{		
2153		if (event_pin->min_threshold < min_event_pin_delay)		
2154		{		
2155		min_event_pin_delay = event_pin->min_threshold;		
2156		if (event_pin->max_threshold > max_event_pin_delay)		
2157		{		
2158		{		
2159		{		
2160		/* Convert the min-max skew to picosec */		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/tmeas.c

DATE 5/23/89  
TIME 6:14:50 pm

PAGE #  
19/150

```

LINE # SOURCE TEXT
2161  lm_tmq_get_sample_ramp_delay(resolution,
2162  (u_char) min_event_pin_delay,
2163  &min_event_pin_delay,
2164  &measurement_error);
2165  min_event_pin_delay += measurement_error;
2166
2167  lm_tmq_get_sample_ramp_delay(resolution,
2168  (u_char) max_event_pin_delay,
2169  &max_event_pin_delay,
2170  &measurement_error);
2171  max_event_pin_delay += measurement_error;
2172
2173  /* Using EARLY SAMPLE mode */
2174  for (i = 0; i < event_pin_count; ++i) {
2175      event_pin = stm_pin_info_table[event_pin_number[i]];
2176
2177      if (resolution == RESOLUTION_05_NS) {
2178          if (event_pin->delay_found == FALSE) {
2179              lm_queue_message(ERROR_MSG, "internal error: event pin: %s of instance: %s delay not found",
2180              def_ptr->pin_table[event_pin_number[i]].pin_name,
2181              instance->device_info_string);
2182              return (FAILURE);
2183          }
2184      } else {
2185          break;
2186      }
2187  }
2188
2189  if (i == event_pin_count) {
2190      /* All of the event pin delays are found. */
2191
2192      for (i = 0; i < event_pin_count; ++i) {
2193          event_pin = stm_pin_info_table[event_pin_number[i]];
2194
2195          /* Save the delays in gbl_eval_min/max_delay[] */
2196          lm_tmq_get_sample_ramp_delay(resolution,
2197          event_pin->min_threshold,
2198          &gbl_eval_min_delay[i],
2199          &measurement_error);
2200          gbl_eval_min_delay[i] += measurement_error;
2201
2202          lm_tmq_get_sample_ramp_delay(resolution,
2203          event_pin->max_threshold,
2204          &gbl_eval_max_delay[i],
2205          &measurement_error);
2206          gbl_eval_max_delay[i] += measurement_error;
2207
2208          /* Calculate the worst case threshold */
2209          if (event_pin->min_threshold < min_event_pin_delay)
2210              min_event_pin_delay = event_pin->min_threshold;
2211          if (event_pin->max_threshold > max_event_pin_delay)
2212              max_event_pin_delay = event_pin->max_threshold;
2213      }
2214
2215      /* Convert the min-max skew to picoseconds */
2216      lm_tmq_get_sample_ramp_delay(resolution,
2217      (u_char) min_event_pin_delay,
2218      &min_event_pin_delay,
2219      &measurement_error);
2220      min_event_pin_delay += measurement_error;
2221
2222      lm_tmq_get_sample_ramp_delay(resolution,
2223      (u_char) max_event_pin_delay,
2224      &max_event_pin_delay,
2225      &measurement_error);
2226      max_event_pin_delay += measurement_error;
2227  } else {
2228      /*
2229       * Not all of the event pin delays are found. Use the delays from
2230       * previous resolution.
2231       */
2232
2233      min_event_pin_delay = MAXINT;
2234      max_event_pin_delay = 0;
2235
2236      for (i = 0; i < event_pin_count; ++i) {
2237          event_pin = stm_pin_info_table[event_pin_number[i]];
2238
2239          if (gbl_eval_min_delay[i] < min_event_pin_delay)
2240              min_event_pin_delay = gbl_eval_min_delay[i];
2241          if (gbl_eval_max_delay[i] > max_event_pin_delay)
2242              max_event_pin_delay = gbl_eval_max_delay[i];
2243      }
2244  }
2245
2246  *comp_eval_min_delay = min_event_pin_delay;
2247  *comp_eval_max_delay = max_event_pin_delay;
2248
2249  return (SUCCESS);
2250
2251
2252

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.cDATE 5/23/89  
TIME 6:14:53 pmPAGE #  
1/151

```

LINE # SOURCE TEXT
1  /* SCSS ID: util.c Rev 3.2, 5/9/89 at 11:02:18 */
2  #include "common.h"
3  #include "device.h"
4  #include "message.h"
5  #include "hardware.h"
6  #include "lm_rd_wr.h"
7  #include "mod_err.h"
8  #include "vvarm.h"
9  #include "seepm.h"
10 #include "lmserver.h"
11 #include "function.h"
12 #include "proctm.h"
13 #include "proctm.h"
14 #include "tmg.h"
15 #include "lm_eff.h"
16 #include "lm_eff.h"
17
18 #ifdef MODELER
19 #include "vrtx.h"
20 #endif
21
22 #define lm_tmg_lane_select(ident_lane) (tmgptr->lane_enable = ident_lane)
23
24 extern LM_HARDWARE_ERROR modeler_error;
25 extern CONFIGURATION_ERRORS config_error;
26
27 extern TMG *tmgptr;
28
29 u_long lm_loop_patterns_count;
30
31 init()
32 {
33     PTRN_BITS *ptrn_bits_ptr;
34     u_short i, j;
35     u_long mask;
36     u_short pin_number;
37     u_char wait_pin_number;
38
39 #ifdef MODELER
40     /* ??? used to substitute VRTX library lm_timer() and lm_delay() */
41     lm_timer();
42 #endif
43
44     profile_init();
45
46     init_config_error();
47
48     /* initialize data structure */
49     for (i = 0; i < MAX_USER_COUNT; ++i) {
50         user_info_array[i] = (USER_INFO *)
51             DCALLOC((unsigned)sizeof(USER_INFO));
52         if (user_info_array[i] == NULL)
53             fatal_alloc_error();
54     }
55
56     /* initialize the dab_list array to NULL */
57     for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i)
58         dab_list[i] = NULL;
59
60     /* initialize misc variables */
61     system_config = NULL;
62
63     for (i = 0; i < MAX_FUNCTION; i++)
64         function_array[i] = process_no_suc;
65
66     function_array[(CREATE_DEF_CMD - 2) / 2] = process_create_def_cmd;
67     function_array[(CREATE_INSTANCE_CMD - 2) / 2] = process_create_instance_cmd;
68     function_array[(CREATE_FAULT_CMD - 2) / 2] = process_create_fault_cmd;
69
70     function_array[(RELEASE_DEF_CMD - 2) / 2] = process_release_def_cmd;
71     function_array[(RELEASE_INSTANCE_CMD - 2) / 2] = process_release_instance_cmd;
72     function_array[(RELEASE_FAULT_CMD - 2) / 2] = process_release_fault_cmd;
73
74     function_array[(EVAL_CMD - 2) / 2] = process_eval_cmd;
75
76     function_array[(SAVE_DEF_CMD - 2) / 2] = process_save_def_cmd;
77     function_array[(SAVE_PTRN_CMD - 2) / 2] = process_save_ptrn_cmd;
78     function_array[(SAVE_PTRN_COUNT_CMD - 2) / 2] = process_save_ptrn_count_cmd;
79     function_array[(RESTORE_INST_CMD - 2) / 2] = process_restore_inst_cmd;
80     function_array[(RESTORE_PTRN_CMD - 2) / 2] = process_restore_ptrn_cmd;
81     function_array[(PTRN_HIST_CMD - 2) / 2] = process_ptrn_hist_cmd;
82
83     function_array[(INQ_MODELER_CMD - 2) / 2] = process_inq_modeler_cmd;
84     function_array[(INQ_USER_LIST_CMD - 2) / 2] = process_inq_user_list_cmd;
85     function_array[(INQ_USER_CMD - 2) / 2] = process_inq_user_cmd;
86     function_array[(INQ_LANE_CMD - 2) / 2] = process_inq_lane_cmd;
87     function_array[(INQ_PAN_CMD - 2) / 2] = process_inq_pan_cmd;
88     function_array[(INQ_PEL_CMD - 2) / 2] = process_inq_pel_cmd;
89     function_array[(INQ_DEVICE_LIST_CMD - 2) / 2] = process_inq_device_list_cmd;
90     function_array[(INQ_DEVICE_NAME_CMD - 2) / 2] = process_inq_device_name_cmd;
91     function_array[(INQ_DEVICE_CMD - 2) / 2] = process_inq_device_cmd;
92     function_array[(INQ_DAB_CMD - 2) / 2] = process_inq_dab_cmd;
93     function_array[(INQ_DAB_LOC_CMD - 2) / 2] = process_inq_dab_loc_cmd;
94     function_array[(INQ_INSTANCE_CMD - 2) / 2] = process_inq_instance_cmd;
95     function_array[(INQ_FAULT_CMD - 2) / 2] = process_inq_fault_cmd;
96
97     function_array[(INQ_AVAIL_PTRN_CMD - 2) / 2] = process_inq_avail_ptrn_cmd;
98
99     function_array[(MEASUREMENT_CMD - 2) / 2] = process_measurement_cmd;
100
101     function_array[(LOOP_PTRN_CMD - 2) / 2] = process_loop_ptrn_cmd;
102
103     function_array[(RESET_INST_CMD - 2) / 2] = process_reset_inst_cmd;
104
105     function_array[(TEST_NETWORK_CMD - 2) / 2] = process_test_network_cmd;
106
107     function_array[(ABORT_CMD - 2) / 2] = process_abort_cmd;
108
109     function_array[(BEGIN_SESSION_CMD - 2) / 2] = process_begin_session_cmd;
110
111     function_array[(LABEL_DAB_CMD - 2) / 2] = process_label_dab_cmd;
112
113     function_array[(EVAL_CONTROL_CMD - 2) / 2] = process_eval_control_cmd;
114
115     function_array[(REBOOT_CMD - 2) / 2] = process_reboot_cmd;
116     function_array[(SHUTDOWN_CMD - 2) / 2] = process_shutdown_cmd;
117
118     function_array[(CHECK_DABDEF_CMD - 2) / 2] = process_check_dabdef_cmd;
119
120

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
2/152

```

LINE # SOURCE TEXT
121 function_array((READ_CMD - 2) / 2) = process_lm_read;
122 function_array((WRITE_CMD - 2) / 2) = process_lm_write;
123
124 function_array((SAVE_DEF_CMD - 2) / 2) = process_save_def_cmd;
125 function_array((PASSWORD_CMD - 2) / 2) = process_password_cmd;
126
127 /* initialize feedback block count array */
128 for (i = 0; i < MAX_LANE_COUNT; ++i) {
129     for (j = 0; j < MAX_FAN_COUNT; ++j) {
130         fb_block_count_array[i][FAN_MAX_ADDR[j] - MAXINT;
131         fb_block_count_array[i].feedback_block_count[j] = 0;
132     }
133 }
134
135 /* initialize the dummy pattern */
136 gbl_dummy_ptr = (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
137                                     (unsigned)sizeof(PTRN_BITS));
138 if (gbl_dummy_ptr == NULL)
139     fatal_alloc_error();
140
141 ptrn_bits_ptr = gbl_dummy_ptr;
142 for (i = 0; i < MAX_UNIT_COUNT; ++i) {
143     set_pel_ctl(ptrn_bits_ptr->ctl, PEL_CTL_SELECT_PEL);
144     ++ptrn_bits_ptr;
145 }
146
147 /* initialize Word Mask Table which is used to convert bit number to
148    = bit mask. */
149 for (i = 0; mask = 1; i < 32; ++i, mask <= 1) {
150     bitso_to_mask[i] = mask;
151 }
152
153 /* initialize global full value structure */
154 allocate_full_value(&gbl_new_sample_value);
155 allocate_full_value(&gbl_steady_state_result);
156 allocate_full_value(&gbl_temp_steady_state_result);
157
158 /* initialize BIT TO LOGIC TABLE */
159 bit_to_logic_table[0x0] = LOGIC_0;
160 bit_to_logic_table[0x1] = LOGIC_1;
161 bit_to_logic_table[0x2] = LOGIC_30;
162 bit_to_logic_table[0x3] = LOGIC_31;
163
164 bit_to_logic_table[0x4] = LOGIC_20;
165 bit_to_logic_table[0x5] = LOGIC_21;
166 bit_to_logic_table[0x6] = LOGIC_20;
167 bit_to_logic_table[0x7] = LOGIC_21;
168
169 bit_to_logic_table[0x8] = LOGIC_U;
170 bit_to_logic_table[0x9] = LOGIC_U;
171 bit_to_logic_table[0xa] = LOGIC_U;
172 bit_to_logic_table[0xb] = LOGIC_U;
173 bit_to_logic_table[0xc] = LOGIC_U;
174 bit_to_logic_table[0xd] = LOGIC_U;
175 bit_to_logic_table[0xe] = LOGIC_U;
176 bit_to_logic_table[0xf] = LOGIC_U;
177
178 /* initialize pin_number to unitno, wordno, bitno offset array */
179 for (pin_number = 0; pin_number < MAX_PIN_COUNT; ++pin_number) {
180     ps_to_short_offset[pin_number].unitno = pin_number / 80;
181     unit_pin_number = pin_number % 80;
182     ps_to_short_offset[pin_number].wordno = (79 - unit_pin_number) / 16;
183     ps_to_short_offset[pin_number].bitno = 15 - (79 - unit_pin_number) % 16;
184 }
185
186 #ifdef DBASE
187 /* initialize data base */
188 db_init();
189 #endif
190
191
192 allocate_full_value(full_value_ptr);
193 FULL_VALUE *full_value_ptr;
194 {
195     full_value_ptr->data =
196         (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
197                             (unsigned)sizeof(PTRN_BITS));
198     if (full_value_ptr->data == NULL)
199         fatal_alloc_error();
200
201     full_value_ptr->hiz =
202         (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
203                             (unsigned)sizeof(PTRN_BITS));
204     if (full_value_ptr->hiz == NULL)
205         fatal_alloc_error();
206
207     full_value_ptr->unknown =
208         (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
209                             (unsigned)sizeof(PTRN_BITS));
210     if (full_value_ptr->unknown == NULL)
211         fatal_alloc_error();
212
213     full_value_ptr->soft =
214         (PTRN_BITS *)DCALLOC((unsigned)MAX_UNIT_COUNT,
215                             (unsigned)sizeof(PTRN_BITS));
216     if (full_value_ptr->soft == NULL)
217         fatal_alloc_error();
218 }
219
220
221 SYSTEM_INFO *
222 new_system_info()
223 {
224     SYSTEM_INFO *temp;
225     u_char i;
226
227     temp = (SYSTEM_INFO *)DCALLOC((unsigned)1,
228                                   (unsigned)sizeof(SYSTEM_INFO));
229     if (temp == NULL)
230         fatal_alloc_error();
231     for (i = 0; i < MAX_LANE_COUNT; ++i)
232         temp->lane[i] = NULL;
233     return(temp);
234 }
235
236
237 LANE_INFO *
238 new_lane_info()
239 {
240

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	3/153

```

LINE #          SOURCE TEXT
241  LANE_INFO *temp;
242  u_char i;
243
244  temp = (LANE_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(LANE_INFO));
245  if (temp == NULL)
246      fatal_alloc_error();
247
248  temp->pac_present = FALSE;
249
250  for (i=0; i < MAX_PAM_COUNT; i++)
251      temp->pam[i] = NULL;
252  for (i=0; i < MAX_SLOT_COUNT; i++)
253      temp->pel[i] = NULL;
254  return(temp);
255
256
257  PEL_INFO *
258  new_pel_info()
259  {
260      PEL_INFO *temp;
261
262      temp = (PEL_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(PEL_INFO));
263      if (temp == NULL)
264          fatal_alloc_error();
265      return(temp);
266  }
267
268
269  DAB_INFO *
270  new_dab_info()
271  {
272      /* creates the DAB info */
273
274      DAB_INFO *temp;
275      u_char i;
276
277      temp = (DAB_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(DAB_INFO));
278      if (temp == NULL)
279          fatal_alloc_error();
280
281      temp->used_as_private = FALSE;
282
283      for (i = 0; i <= MAX_SEGMENT_PER_DEVICE; ++i)
284          temp->segment[i] = NULL;
285
286      return(temp);
287  }
288
289  rls_dab_info(dab_ptr)
290  DAB_INFO *dab_ptr;
291  {
292      u_char i;
293
294      for (i = 0; i <= MAX_SEGMENT_PER_DEVICE; ++i)
295          if (dab_ptr->segment[i] != NULL)
296              DFREE((char *)dab_ptr->segment[i]);
297
298      DFREE((char *)dab_ptr);
299  }
300
301  enter_dab_info(loc_dab_list, dab_ptr, laneso, slotso)
302  DAB_INFO *loc_dab_list[];
303  DAB_INFO *dab_ptr;
304  u_short laneso;
305  u_short slotso;
306  {
307
308      u_char index;
309
310      index = laneso * MAX_SLOT_COUNT + slotso;
311
312      DPRINTF(("inside enter_dab_info, name: %s laneso: %d slotso: %d index: %d\n",
313          dab_ptr->part_name, laneso, slotso, index));
314
315      loc_dab_list[index] = dab_ptr;
316  }
317
318
319  find_dab(loc_dab_list, name, device_type)
320  DAB_INFO *loc_dab_list[];
321  char *name;
322  u_char device_type; /* PRIVATE or PUBLIC */
323  {
324      /*
325       * Find the dab name in the device list.
326       * Return the index to dab_list if the name is found otherwise return -1
327       */
328
329      DAB_INFO *dab_ptr;
330      u_short i;
331
332      DPRINTF(("inside find_dab\n"));
333
334      for (i = 0; i < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++i) {
335          dab_ptr = loc_dab_list[i];
336          if (dab_ptr != NULL)
337              if (strcmp(dab_ptr->part_name, name) == 0) {
338                  if (device_type == PRIVATE) {
339                      if ((dab_ptr->used_as_private == FALSE) ||
340                          ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0))
341                      return(i);
342                  }
343                  else {
344                      if (dab_ptr->used_as_private == FALSE)
345                          return(i);
346                  }
347              }
348      }
349
350      return(-1);
351  }
352
353
354  PAM_INFO *
355  new_pam_info()
356  {
357      PAM_INFO *temp;
358
359      temp = (PAM_INFO *)DCALLOC((unsigned)1, (unsigned)sizeof(PAM_INFO));
360      if (temp == NULL)

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
4/154

```

LINE #          SOURCE TEXT
361      fatal_alloc_error();
362      return(temp);
363  }
364  }
365  SEGMENT_EL
366  new_segment()
367  {
368      SEGMENT_EL *temp;
369      temp = (SEGMENT_EL *)DCALLOC((unsigned)1, (unsigned)sizeof(SEGMENT_EL));
370      if (temp == NULL)
371          fatal_alloc_error();
372      return(temp);
373  }
374  }
375  }
376  }
377  }
378  set_last_in_lane(dab_ptr)
379  DAB_INFO *dab_ptr;
380  {
381      /* setup the following field in dab_info:
382       *   last_in_lane
383       *   lane_used field
384       *   lane_count
385       *   unit_count_per_lane
386       *   dummy_ptrn
387       *   ident_lane
388       */
389
390      char last_lane_found[MAX_LANE_COUNT];
391      char unit_count[MAX_LANE_COUNT];
392      u_char laneso;
393      u_char i;
394      u_char ident_lane = 0;
395      u_char count;
396      u_char any_used_slotso_on_lane[MAX_LANE_COUNT];
397
398      for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
399          last_lane_found[laneso] = 0;
400          unit_count[laneso] = 0;
401      }
402
403      count = dab_ptr->unit_count;
404      for (i = 0; i < count; ++i) {
405          laneso = dab_ptr->unit_location[i].lane_no;
406          dab_ptr->lane_used[laneso] = 1;
407          if (last_lane_found[laneso] == 0) {
408              dab_ptr->unit_location[i].last_in_lane = 1;
409              last_lane_found[laneso] = 1;
410              ident_lane |= 1 << laneso;
411          }
412          any_used_slotso_on_lane[laneso] = dab_ptr->unit_location[i].slot_no;
413          ++unit_count[laneso];
414      }
415      dab_ptr->ident_lane = ident_lane;
416
417      for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
418          if (dab_ptr->lane_used[laneso]) {
419              ++dab_ptr->lane_count;
420              if (unit_count[laneso] > dab_ptr->unit_count_per_lane)
421                  dab_ptr->unit_count_per_lane = unit_count[laneso];
422              dab_ptr->dummy_ptrn[laneso].word[2] =
423                  DUMMY_CTL_WORD(any_used_slotso_on_lane[laneso]);
424          }
425      }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

5/155

```

LINE # SOURCE TEXT
481 inst_ptr = (INSTANCE_INFO *)temp->instance[i];
482 for (j = 0; j < (INST_TABLE_SIZE - 1); ++j) {
483     temp->instance[j] = (INSTANCE_INFO *)inst_ptr;
484     ++inst_ptr;
485 }
486 temp->instance[j] = NULL;
487 temp->free_inst_id = temp->instance;
488
489 /* create the definition table for this user */
490 temp->def_table_size = DEF_TABLE_SIZE;
491 temp->definition = (DEVICE_SPEC *)
492     EMALLOC((unsigned)(DEF_TABLE_SIZE * sizeof(DEVICE_SPEC)));
493 if (temp->definition == NULL)
494     return(FAILURE);
495
496 def_ptr = (DEVICE_SPEC *)temp->definition[i];
497 for (j = 0; j < (DEF_TABLE_SIZE - 1); ++j) {
498     temp->definition[j] = (DEVICE_SPEC *)def_ptr;
499     ++def_ptr;
500 }
501
502 temp->definition[j] = NULL;
503 temp->free_def_id = temp->definition;
504
505 return(SUCCESS);
506 }
507
508 abort_user(user)
509 USER_INFO *user;
510 {
511     DAB_INFO *dab_ptr;
512     DEVICE_SPEC *definition;
513     INSTANCE_INFO *instance;
514     EXTRA_DEVICE_SPEC *extra_def_ptr;
515     u_long pattern_count;
516     u_long pattern_count1;
517     u_long longest_pattern_seq;
518     u_long cur_pattern_count;
519     u_short def_count;
520     u_short inst_count;
521     u_short fault_count;
522     i;
523     u_char lane_count;
524     u_char lane;
525
526     DPRINTF(("inside abort_user\n"));
527
528     if (user->active == FALSE) {
529         lm_queue_message(ERROR_MSG, "internal error: attempt to release unused user");
530         return;
531     }
532
533     if (user->save_buffer != NULL) {
534         DPRINTF(user->save_buffer);
535     }
536
537     pattern_count1 = 0;
538     pattern_count = 0;
539     longest_pattern_seq = 0;
540     def_count = 0;
541     inst_count = 0;
542     fault_count = 0;
543     lane_count = 0;
544
545     /* Free instances */
546     for (i = 0; i < user->inst_table_size; ++i) {
547         instance = user->instance[i];
548
549         /* check if it's a valid instance */
550         if (BOGUS_INSTANCE(user, instance))
551             continue;
552
553         extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;
554
555         for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
556             if (extra_def_ptr->lane_used & 1 << lane)
557                 ++lane_count;
558         }
559
560         cur_pattern_count = instance->pattern_count * lane_count;
561
562         add_4((u_long)pattern_count, (u_long)0, cur_pattern_count);
563
564         if (cur_pattern_count > longest_pattern_seq)
565             longest_pattern_seq = cur_pattern_count;
566
567         return_all_ptr_block(instance);
568
569         if ((char)instance->dab_info_index != -1) {
570             dab_ptr = dab_list(instance->dab_info_index);
571
572             if (instance->is_fault == TRUE) {
573                 --dab_ptr->act_var_count;
574                 ++fault_count;
575             }
576             else {
577                 --dab_ptr->act_inst_count;
578                 ++inst_count;
579             }
580
581             if (dab_ptr->used_as_private == TRUE) {
582                 dab_ptr->used_as_private = FALSE;
583                 set_private_mode(dab_ptr, FALSE);
584             }
585
586             if ((dab_ptr->act_inst_count + dab_ptr->act_var_count) == 0)
587                 turn_off_in_use(dab_ptr);
588         }
589
590         xis_instance(user, i);
591     }
592
593     /* Free definitions */
594     for (i = 0; i < user->def_table_size; ++i) {
595         definition = user->definition[i];
596
597         /* check if it's a valid definition */
598         if (BOGUS_DEFINITION(user, definition))
599             continue;
600     }

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	6/156

```

LINE #          SOURCE TEXT
601      ++def_count,
602      rls_definition(user, 1);
603  }
604
605  /* free the definition and instance tables */
606  DFREE((char *)user->definition);
607  DFREE((char *)user->instance);
608
609  user->active = FALSE;
610
611  write_user_stat(pattern_count, pattern_count1, longest_pattern_seq,
612                def_count, inst_count, fault_count);
613
614  DPRINTF(("exiting about user\n"));
615
616  }
617
618  new_and_link_definition(user, def_ptr, def_id_table_index)
619  USER_INFO *user;
620  DEVICE_SPEC *def_ptr;
621  u_short def_id_table_index;
622  {
623
624      DEVICE_SPEC **slot_ptr;
625
626      DPRINTF(("inside new_and_link_definition\n"));
627
628      if (user->free_def_id == NULL)
629          if (extend_def_table(user) == FAILURE)
630              return(FAILURE);
631
632      slot_ptr = user->free_def_id;
633
634      *def_id_table_index = (u_short)
635          (((u_long)slot_ptr - (u_long)user->definition) / sizeof(DEVICE_SPEC));
636
637      user->free_def_id = (DEVICE_SPEC **)slot_ptr;
638      *slot_ptr = def_ptr;
639
640      def_ptr->extra_data = (char *)DCALLOC((unsigned)1,
641                                           (unsigned)sizeof(EXTRA_DEVICE_SPEC));
642      if (def_ptr->extra_data == NULL)
643          return(FAILURE);
644      ((EXTRA_DEVICE_SPEC *)def_ptr->extra_data)->dab_ok = TRUE;
645
646      return(SUCCESS);
647  }
648
649  allocate_initial_block(instance)
650  INSTANCE_INFO *instance;
651  {
652      /* Allocate a new block in each lane used by this instance.
653       * Setup the following fields:
654       *   LANE_ADDR_INFO.prev_max_addr
655       *   LANE_ADDR_INFO.last_unit_addr
656       *   LANE_ADDR_INFO.new_block_addr
657       *   INSTANCE_INFO.pattern_count
658       *   INSTANCE_INFO.unit_addr
659       */
660
661      DAB_INFO *dab_ptr;
662      LANE_ADDR_INFO *lane_ptr;
663      u_long ptrs_addr_to_assign(MAX_LANE_COUNT);
664      u_long temp_unit_addr;
665      u_char unit_processed(MAX_LANE_COUNT);
666      u_char unit_count(MAX_LANE_COUNT);
667      u_char unitno;
668      u_char i;
669      u_char dummy_unit_offset;
670      u_char junk;
671
672      DPRINTF(("inside allocate_initial_block\n"));
673
674      dab_ptr = dab_list(instance->dab_info_index);
675
676      for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {
677          lane_ptr = (LANE_ADDR_INFO *)instance->lane_addr[lane];
678          if (dab_ptr->lane_used[lane]) {
679              lane_ptr->prev_max_addr = lane_ptr->max_addr;
680
681              if (new_block(lane_ptr->max_addr,
682                          &junk,
683                          PATTERNS_BLOCK,
684                          lane) == FAILURE) {
685                  lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory");
686                  instance->failed_to_alloc_ptrs = TRUE;
687                  return(FAILURE);
688              }
689
690              lane_ptr->max_addr += BLOCK_ADDR_INC;
691
692              lane_ptr->last_unit_addr = lane_ptr->max_addr - BLOCK_ADDR_INC +
693                  (dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC;
694
695              lane_ptr->new_block_addr = 0;
696
697              ptrs_addr_to_assign[lane] = lane_ptr->max_addr - BLOCK_ADDR_INC;
698          }
699          else {
700              ptrs_addr_to_assign[lane] = 0;
701          }
702      }
703
704      /* Initialize unit_count[] and unit_processed[] to be used later */
705      unit_count[lane] = 0;
706      unit_processed[lane] = 0;
707
708      /* Calculate the number of actual units in each lane */
709      for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno)
710          ++unit_count[dab_ptr->unit_location[unitno].lane_no];
711
712      /* Assign the addresses for the dummy patterns */
713      dummy_unit_offset = dab_ptr->unit_count + 1;
714      temp_unit_addr = instance->unit_addr[0][0];
715      instance->cur_unit_addr_index = 0;
716  }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
7/157

```

LINE # SOURCE TEXT
721 for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
722     if (dab_ptr->lane_used[lane0]) {
723         for (i = 0; i < dab_ptr->unit_count_per_lane -
724             unit_count[lane0]; ++i) {
725             temp_unit_addr[dummy_unit_offset++] =
726                 ptrs_addr_to_assign[lane0];
727             ptrs_addr_to_assign[lane0] += PTRN_ADDR_INC;
728         }
729     }
730     ptrs_addr_to_assign[lane0] += PTRN_ADDR_INC;
731 }
732
733 /* Assign the addresses for the actual units */
734 for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
735     temp_unit_addr[unitno] =
736         ptrs_addr_to_assign[dab_ptr->unit_location[unitno].lane_no];
737     ptrs_addr_to_assign[dab_ptr->unit_location[unitno].lane_no] +=
738         PTRN_ADDR_INC;
739 }
740
741 instance->pattern_count += dab_ptr->unit_count_per_lane;
742
743 return(SUCCESS);
744
745 load_dummy_pattern(instance)
746 INSTANCE_INFO *instance;
747 {
748     /* This routine builds some dummy patterns IF necessary so that a proper
749        * branch can be made to the feedback sequence.
750        */
751
752     DAB_INFO *dab_ptr;
753     u_long temp_unit_addr;
754     u_short ptrs_count;
755     u_short dummy_ptr_count;
756     i;
757     u_short quiet_patterns;
758     u_char total_unit;
759     u_char bits_per_pin = 1;
760     u_char unitno;
761
762     dab_ptr = dab_list(instance->dab_info_index);
763
764     quiet_patterns = 4 + 3 * (dab_ptr->unit_count_per_lane * bits_per_pin);
765
766     ptrs_count = quiet_patterns / dab_ptr->unit_count_per_lane;
767     if (quiet_patterns % dab_ptr->unit_count_per_lane != 0)
768         ++ptrs_count;
769
770     ptrs_count += (instance->definition->pre_seq_len +
771         sizeof(PREAMBLE) / sizeof(PTRN_BITS)) *
772         dab_ptr->unit_count_per_lane;
773
774     dummy_ptr_count = 0;
775
776     if (ptrs_count % PTRN_PER_BLOCK != BRANCH_LATENCY) {
777         dummy_ptr_count = (BRANCH_LATENCY - (ptrs_count % PTRN_PER_BLOCK) +
778             dab_ptr->unit_count_per_lane) /
779             dab_ptr->unit_count_per_lane;
780     }
781
782     if (((ptrs_count + dummy_ptr_count) % dab_ptr->unit_count_per_lane) != 1) {
783         /* There will be an odd number of unit patterns before the feedback
784            * sequence. This means that the last unit pattern before the
785            * feedback sequence will be on an even pattern address. The branch
786            * command has to be specified 24 patterns before this last unit pattern,
787            * so it will also be on even pattern address. Since the branch can only
788            * be made from an odd pattern address, therefore we need to add ONE
789            * UNIT of dummy patterns on each lane to make a proper branch.
790            */
791         adjust_patterns_addr(instance);
792
793         write_patterns_unit(instance, dab_ptr->dummy_ptr_count);
794
795         /* Increment the address of each unit by PTRN_ADDR_INC. Also set the
796            * LANE_ADDR_INFO.last_unit_addr correctly because it was modified by
797            * adjust_patterns_addr(). We don't have to worry about going over
798            * current block max address because this is the first pattern in
799            * the block.
800            */
801         total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;
802         temp_unit_addr = instance->unit_addr(instance->cur_unit_addr_index)[0];
803         for (unitno = 0; unitno < total_unit; ++unitno) {
804             temp_unit_addr[unitno] += PTRN_ADDR_INC;
805             if (dab_ptr->unit_location[unitno].last_in_lane)
806                 instance->lane_addr[dab_ptr->unit_location[unitno].lane_no].
807                     last_unit_addr = temp_unit_addr[unitno];
808         }
809
810         /* The pattern_count was subtracted in adjust_patterns_addr() */
811         instance->pattern_count += dab_ptr->unit_count_per_lane;
812     }
813
814     for (i = 0; i < dummy_ptr_count; ++i) {
815         write_patterns(instance,
816             instance->unit_addr(instance->cur_unit_addr_index)[0],
817             gbl_dummy_ptr);
818         if (grow_patterns(instance) == FAILURE)
819             return(FAILURE);
820     }
821
822     return(SUCCESS);
823 }
824
825 load_dummy_patterns2(instance)
826 INSTANCE_INFO *instance;
827 {
828     /* This routine builds some dummy patterns so that the feedback signal
829        * is quiet before we branch to the feedback sequence.
830        */
831
832     DAB_INFO *dab_ptr;
833     u_short dummy_ptr_count;
834     u_short i;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
8/158

```

LINE # SOURCE TEXT
841 u_short quiet_patterns;
842 u_char bits_per_pis = 1;
843
844 dab_ptr = dab_list(instance->dab_info_index);
845
846 quiet_patterns = 4 + 3 * (dab_ptr->unit_count_per_lane * DAB_BITS_PER_PIS);
847
848 dummy_ptr_count = quiet_patterns / dab_ptr->unit_count_per_lane;
849 if (quiet_patterns % dab_ptr->unit_count_per_lane != 0)
850 ++dummy_ptr_count;
851
852 for (i = 0; i < dummy_ptr_count; ++i) {
853     if (grow_patterns(instance) == FAILURE)
854         return(FAILURE);
855
856     write_patterns(instance,
857                     instance->unit_addr(instance->cur_unit_addr_index)[0],
858                     &dummy_ptr_count);
859 }
860
861 return(SUCCESS);
862
863 }
864
865 allocate_feedback_block(instance)
866 INSTANCE_INFO *instance;
867 {
868     DAB_INFO *dab_ptr;
869     u_long addr_inc;
870     u_long temp_unit_addr;
871     u_char lane_no;
872     u_char unitno;
873
874     dab_ptr = dab_list(instance->dab_info_index);
875
876     /* Allocate feedback blocks on each lane used */
877     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
878         if (dab_ptr->lane_used[lane_no]) {
879             if (new_block(instance->fb_block_addr[lane_no],
880                         instance->fb_block_size[lane_no],
881                         FB_BLOCK_SIZE,
882                         lane_no) == FAILURE) {
883                 instance->failed_to_alloc_ptrn = TRUE;
884                 lm_queue_message(ERROR_MSG, "out of Fast Pattern Memory for feedback");
885                 return(FAILURE);
886             }
887         }
888     }
889
890     /* Calculate how much each unit address has to be incremented to get to
891     * a new block. This number is the same for each lane since up to this
892     * point the patterns grow at the same rate on each lane.
893     */
894     lane_no = dab_ptr->unit_location[0].lane_no;
895     addr_inc = instance->lane_addr[lane_no].max_addr -
896               (instance->lane_addr[lane_no].last_unit_addr -
897                (dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC);
898
899     for (lane_no = 0; lane_no < MAX_LANE_COUNT; ++lane_no) {
900         if (dab_ptr->lane_used[lane_no]) {
901             /* Set the branch command and enable feedback */
902             set_branch(instance->lane_addr[lane_no].last_unit_addr,
903                       instance->fb_block_addr[lane_no]);
904
905             /* Adjust PREV_MAX_ADDR and MAX_ADDR */
906             instance->lane_addr[lane_no].prev_max_addr =
907                 instance->lane_addr[lane_no].max_addr;
908
909             instance->lane_addr[lane_no].max_addr =
910                 instance->fb_block_addr[lane_no] +
911                 instance->fb_block_size[lane_no] * BLOCK_ADDR_INC;
912         }
913     }
914
915     temp_unit_addr = instance->unit_addr(instance->cur_unit_addr_index)[0];
916     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
917         lane_no = dab_ptr->unit_location[unitno].lane_no;
918         temp_unit_addr[unitno] = instance->fb_block_addr[lane_no] +
919                                 temp_unit_addr[unitno] + addr_inc -
920                                 instance->lane_addr[lane_no].prev_max_addr;
921     }
922
923     return(SUCCESS);
924 }
925
926 set_ptrn_bit(ptrn_ptr, wordno, bitno)
927 PTRN_BITS *ptrn_ptr;
928 u_char wordno;
929 u_char bitno;
930 {
931     ptrn_ptr->word[wordno] |= (u_short)bitno_to_mask(bitno);
932 }
933
934 reset_ptrn_bit(ptrn_ptr, wordno, bitno)
935 PTRN_BITS *ptrn_ptr;
936 u_char wordno;
937 u_char bitno;
938 {
939     ptrn_ptr->word[wordno] &= (u_short)bitno_to_mask(bitno);
940 }
941
942 toggle_ptrn_bit(ptrn_ptr, wordno, bitno)
943 PTRN_BITS *ptrn_ptr;
944 u_char wordno;
945 u_char bitno;
946 {
947     ptrn_ptr->word[wordno] ^= (u_short)bitno_to_mask(bitno);
948 }
949
950 read_ptrn_bit(ptrn_ptr, wordno, bitno)
951 PTRN_BITS *ptrn_ptr;
952 u_char wordno;
953 u_char bitno;
954 {
955     return(ptrn_ptr->word[wordno] & (u_short)bitno_to_mask(bitno));
956 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 9/159
LINE #		SOURCE TEXT		
961		{		
962		return((ptrn_ptr->word[wordno] & (u_short)bitno_to_mask(bitno))    0);		
963		}		
964		set_pin_value(full_value_ptr, unitno, wordno, bitno, pin_value)		
965		FULL_VALUE *full_value_ptr;		
966		u_char unitno;		
967		u_char wordno;		
968		u_char bitno;		
969		u_char pin_value;		
970		{		
971		{		
972		switch (pin_value) {		
973		case LOGIC_0:		
974		reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno);		
975		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
976		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
977		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
978		reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno);		
979		break;		
980		case LOGIC_1:		
981		set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno);		
982		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
983		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
984		reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno);		
985		break;		
986		case LOGIC_20:		
987		reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno);		
988		set_ptrn_bit (full_value_ptr->hiz [unitno], wordno, bitno);		
989		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
990		reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno);		
991		break;		
992		case LOGIC_21:		
993		set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno);		
994		set_ptrn_bit (full_value_ptr->hiz [unitno], wordno, bitno);		
995		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
996		reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno);		
997		break;		
998		case LOGIC_U:		
999		reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno);		
1000		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
1001		set_ptrn_bit (full_value_ptr->unknown [unitno], wordno, bitno);		
1002		reset_ptrn_bit(full_value_ptr->soft [unitno], wordno, bitno);		
1003		break;		
1004		case LOGIC_20:		
1005		reset_ptrn_bit(full_value_ptr->data [unitno], wordno, bitno);		
1006		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
1007		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
1008		set_ptrn_bit (full_value_ptr->soft [unitno], wordno, bitno);		
1009		break;		
1010		case LOGIC_21:		
1011		set_ptrn_bit (full_value_ptr->data [unitno], wordno, bitno);		
1012		reset_ptrn_bit(full_value_ptr->hiz [unitno], wordno, bitno);		
1013		reset_ptrn_bit(full_value_ptr->unknown [unitno], wordno, bitno);		
1014		set_ptrn_bit (full_value_ptr->soft [unitno], wordno, bitno);		
1015		break;		
1016		default:		
1017		break;		
1018		}		
1019		}		
1020		read_pin_value(full_value_ptr, unitno, wordno, bitno)		
1021		FULL_VALUE *full_value_ptr;		
1022		u_char unitno;		
1023		u_char wordno;		
1024		u_char bitno;		
1025		{		
1026		u_char index;		
1027		{		
1028		index = read_ptrn_bit(full_value_ptr->		
1029		unknown[unitno], wordno, bitno) << 3		
1030		read_ptrn_bit(full_value_ptr->		
1031		hiz[unitno], wordno, bitno) << 2		
1032		read_ptrn_bit(full_value_ptr->		
1033		soft[unitno], wordno, bitno) << 1		
1034		read_ptrn_bit(full_value_ptr->		
1035		data[unitno], wordno, bitno);		
1036		return(bit_to_logic_table[index]);		
1037		}		
1038		}		
1039		}		
1040		/*		
1041		The following routines are cousins to the above routines. The difference		
1042		is that the wordno and bitno offsets are longword offsets.		
1043		*/		
1044		{		
1045		set_ptrn_bit_long(ptrn_ptr, wordno, bitno)		
1046		PTRN_BITS_LONGWORD *ptrn_ptr;		
1047		u_char wordno;		
1048		u_char bitno;		
1049		{		
1050		ptrn_ptr->word[wordno]  = bitno_to_mask(bitno);		
1051		}		
1052		{		
1053		reset_ptrn_bit_long(ptrn_ptr, wordno, bitno)		
1054		PTRN_BITS_LONGWORD *ptrn_ptr;		
1055		u_char wordno;		
1056		u_char bitno;		
1057		{		
1058		ptrn_ptr->word[wordno] ^= bitno_to_mask(bitno);		
1059		}		
1060		{		
1061		toggle_ptrn_bit_long(ptrn_ptr, wordno, bitno)		
1062		PTRN_BITS_LONGWORD *ptrn_ptr;		
1063		u_char wordno;		
1064		u_char bitno;		
1065		{		
1066		ptrn_ptr->word[wordno] ^= bitno_to_mask(bitno);		
1067		}		
1068		{		
1069		read_ptrn_bit_long(ptrn_ptr, wordno, bitno)		
1070		PTRN_BITS_LONGWORD *ptrn_ptr;		
1071		u_char wordno;		
1072		u_char bitno;		
1073		{		
1074		return((ptrn_ptr->word[wordno] & bitno_to_mask(bitno))    0);		
1075		}		
1076		{		
1077		set_pin_value_long(full_value_ptr, unitno, wordno, bitno, pin_value)		
1078		FULL_VALUE *full_value_ptr;		
1079		u_char unitno;		
1080		{		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89 PAGE #  
TIME 6:14:53 pm 10/160

```

LINE #          SOURCE TEXT
1081 u_char      wordao,
1082 u_char      bitao,
1083 u_char      pin_value;
1084 {
1085
1086     switch (pin_value) {
1087     case LOGIC_0:
1088         reset_ptrn_bit_logg(&full_value_ptr->data    [unitao], wordao, bitao);
1089         reset_ptrn_bit_logg(&full_value_ptr->hiz      [unitao], wordao, bitao);
1090         reset_ptrn_bit_logg(&full_value_ptr->unknowns [unitao], wordao, bitao);
1091         reset_ptrn_bit_logg(&full_value_ptr->soft     [unitao], wordao, bitao);
1092         break;
1093     case LOGIC_1:
1094         set_ptrn_bit_logg (&full_value_ptr->data    [unitao], wordao, bitao);
1095         set_ptrn_bit_logg (&full_value_ptr->hiz      [unitao], wordao, bitao);
1096         set_ptrn_bit_logg (&full_value_ptr->unknowns [unitao], wordao, bitao);
1097         set_ptrn_bit_logg (&full_value_ptr->soft     [unitao], wordao, bitao);
1098         break;
1099     case LOGIC_20:
1100         reset_ptrn_bit_logg(&full_value_ptr->data    [unitao], wordao, bitao);
1101         set_ptrn_bit_logg (&full_value_ptr->hiz      [unitao], wordao, bitao);
1102         reset_ptrn_bit_logg(&full_value_ptr->unknowns [unitao], wordao, bitao);
1103         reset_ptrn_bit_logg(&full_value_ptr->soft     [unitao], wordao, bitao);
1104         break;
1105     case LOGIC_21:
1106         set_ptrn_bit_logg (&full_value_ptr->data    [unitao], wordao, bitao);
1107         set_ptrn_bit_logg (&full_value_ptr->hiz      [unitao], wordao, bitao);
1108         reset_ptrn_bit_logg(&full_value_ptr->unknowns [unitao], wordao, bitao);
1109         reset_ptrn_bit_logg(&full_value_ptr->soft     [unitao], wordao, bitao);
1110         break;
1111     case LOGIC_U:
1112         reset_ptrn_bit_logg(&full_value_ptr->data    [unitao], wordao, bitao);
1113         reset_ptrn_bit_logg(&full_value_ptr->hiz      [unitao], wordao, bitao);
1114         set_ptrn_bit_logg (&full_value_ptr->unknowns [unitao], wordao, bitao);
1115         reset_ptrn_bit_logg(&full_value_ptr->soft     [unitao], wordao, bitao);
1116         break;
1117     case LOGIC_30:
1118         reset_ptrn_bit_logg(&full_value_ptr->data    [unitao], wordao, bitao);
1119         reset_ptrn_bit_logg(&full_value_ptr->hiz      [unitao], wordao, bitao);
1120         reset_ptrn_bit_logg(&full_value_ptr->unknowns [unitao], wordao, bitao);
1121         set_ptrn_bit_logg (&full_value_ptr->soft     [unitao], wordao, bitao);
1122         break;
1123     case LOGIC_31:
1124         set_ptrn_bit_logg (&full_value_ptr->data    [unitao], wordao, bitao);
1125         reset_ptrn_bit_logg(&full_value_ptr->hiz      [unitao], wordao, bitao);
1126         reset_ptrn_bit_logg(&full_value_ptr->unknowns [unitao], wordao, bitao);
1127         set_ptrn_bit_logg (&full_value_ptr->soft     [unitao], wordao, bitao);
1128         break;
1129     default:
1130         break;
1131     }
1132
1133     read_pin_value_logg(full_value_ptr, unitao, wordao, bitao)
1134     FULL_VALUE = full_value_ptr;
1135     u_char      unitao;
1136     u_char      wordao;
1137     u_char      bitao;
1138     {
1139         u_char index;
1140
1141         index = read_ptrn_bit_logg(&full_value_ptr->
1142             unknowns[unitao], wordao, bitao) << 3 |
1143             read_ptrn_bit_logg(&full_value_ptr->
1144                 hiz[unitao], wordao, bitao) << 2 |
1145             read_ptrn_bit_logg(&full_value_ptr->
1146                 soft[unitao], wordao, bitao) << 1 |
1147             read_ptrn_bit_logg(&full_value_ptr->
1148                 data[unitao], wordao, bitao);
1149
1150         return(bit_to_logic_table[index]);
1151     }
1152
1153     set_pel_ctl(word, value)
1154     u_short *word;
1155     u_char value;
1156     {
1157         *word = (*word & PEL_CTL_MASK) | (value << PEL_CTL_SHIFT);
1158     }
1159
1160     set_pel_user_bit(word)
1161     u_short *word;
1162     {
1163         *word |= PEL_USER_MASK;
1164     }
1165
1166     clear_pel_user_bit(word)
1167     u_short *word;
1168     {
1169         *word &= PEL_USER_MASK;
1170     }
1171
1172     set_pel_sel(word, value)
1173     u_short *word;
1174     u_char value;
1175     {
1176         *word = (*word & PEL_SEL_MASK) | (value << PEL_SEL_SHIFT);
1177     }
1178
1179     set_pel_br(word, value)
1180     u_short *word;
1181     u_char value;
1182     {
1183         *word = (*word & PEL_BRANCH_MASK) | (value << PEL_BRANCH_SHIFT);
1184     }
1185
1186     new_block(address, count, type, laneso)
1187     u_long *address;
1188     u_char *count;
1189     u_char laneso;
1190     u_char type;
1191     {
1192         /* if type == PATTERN_BLOCK then allocate 1 block on specified lane
1193          * if type == FEEDBACK_BLOCK then allocate n blocks on specified lane
1194          * Returns:
1195          *   SUCCESS
1196          *   the starting addr of the block, and count if can allocate block
1197          *   FAILURE
1198          *   if cannot allocate block
1199         */
1200     }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89 PAGE #  
TIME 6:14:53 pm 11/161

```

1201  * Note make sure when allocating feedback blocks, that the starting block
1202  * is on the x block boundary (where x is the number of blocks required
1203  * on the PAM type).
1204  */
1205
1206  u_long    temp;
1207  u_long    temp2;
1208  u_long    link_table_addr;
1209  u_long    value;
1210  u_long    value2;
1211  u_long    pam_max_addr;
1212  u_short   block_no;
1213  u_char    i;
1214  u_char    panno;
1215  u_char    needed;
1216
1217  *address = NULL;
1218
1219  switch (type) {
1220  case PATTERN_BLOCK:
1221    if (free_block_list[lansno] == NULL)
1222      return(FAILURE);
1223
1224    temp = free_block_list[lansno];
1225
1226    DPRINTF(("New block addr: %8X\n", temp));
1227
1228    link_table_addr = ptol(temp);
1229
1230    DPRINTF(("link_table_addr: %8X\n", link_table_addr));
1231
1232    write_loc_long((u_long *)link_table_addr, (u_long)SEQ_END_BLOCK_FLAG);
1233
1234    free_block_list[lansno] = read_loc_long((u_long *)temp);
1235
1236    DPRINTF(("addr of prev link: %8X\n", free_block_list[lansno] + 4));
1237    write_loc_long((u_long *)(free_block_list[lansno] + 4), (u_long)NULL);
1238
1239    /*
1240    free_block_list[lansno] = temp->next;
1241    temp->prev = NULL;
1242    */
1243
1244    *address = (u_long)temp;
1245    *count = 1;
1246
1247    break;
1248  case FEEDBACK_BLOCK:
1249    if (free_block_list[lansno] == NULL)
1250      return(FAILURE);
1251
1252    temp = free_block_list[lansno];
1253    while (temp != NULL) {
1254      for (panno = 0; panno < MAX_PAM_COUNT; ++panno) {
1255        if ((u_long)temp < fb_block_count_array[lansno].
1256            PAM_max_addr[panno])
1257          break;
1258      }
1259
1260      needed = fb_block_count_array[lansno].
1261              feedback_block_count[panno];
1262
1263      PAM_max_addr = fb_block_count_array[lansno].
1264                  PAM_max_addr[panno];
1265
1266      /* check if temp is in "needed" boundary */
1267      block_no = ptob(temp);
1268
1269      if (! (block_no & needed)) {
1270        /* Temp is pointing to a block which is on "needed" boundary.
1271        * Now check if ("needed" - 1) blocks after temp are free.
1272        */
1273
1274        temp2 = temp + BLOCK_ADDR_INC;
1275        for (i = 0; i < (needed - 1); ++i) {
1276          /* break from the for loop if temp2 points to an address
1277          * outside the current PAM address range.
1278          */
1279          if ((u_long)temp2 >= PAM_max_addr)
1280            break;
1281
1282          link_table_addr = ptol(temp2);
1283
1284          value = read_loc_long((u_long *)link_table_addr) &
1285                  BLOCK_NUMBER_MASK;
1286
1287          if (value != FREE_BLOCK_FLAG)
1288            break; /* break from the for loop */
1289
1290          temp2 += BLOCK_ADDR_INC;
1291        }
1292
1293        if (i == (needed - 1))
1294          break; /* break from the while loop */
1295      }
1296
1297      temp = read_loc_long((u_long *)temp);
1298    }
1299
1300    if (temp == NULL)
1301      return(FAILURE);
1302
1303    /* We found usable feedback blocks */
1304    *address = (u_long)temp;
1305    *count = needed;
1306
1307    /* Take off "needed" blocks from the free list starting at temp. */
1308    for (i = 0; i < needed; ++i) {
1309      DPRINTF(("New FB block addr: %8X\n", temp));
1310
1311      link_table_addr = ptol(temp);
1312
1313      write_loc_long((u_long *)link_table_addr, (u_long)SEQ_END_BLOCK_FLAG);
1314
1315      value = read_loc_long((u_long *)temp + 4);
1316    }
1317
1318  }
1319
1320

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
12/162

```

1321     if (value == NULL) {
1322         free_block_list(laneno) = read_loc_long((u_long *)temp);
1323     }
1324     else {
1325         value = read_loc_long((u_long *)temp);
1326         value2 = read_loc_long((u_long *)temp + 4);
1327         write_loc_long((u_long *)value2, value);
1328     }
1329
1330     value = read_loc_long((u_long *)temp + 4);
1331     value2 = read_loc_long((u_long *)temp);
1332     write_loc_long((u_long *)value2 + 4, value);
1333
1334     temp = read_loc_long((u_long *)temp);
1335
1336     /*
1337     if (temp->prev == NULL)
1338         free_block_list(laneno) = temp->next;
1339     else
1340         temp->prev->next = temp->next;
1341
1342     temp->next->prev = temp->prev;
1343     temp = temp->next;
1344     */
1345 }
1346
1347 break;
1348 default:
1349     return(FAILURE);
1350 }
1351
1352 return(SUCCESS);
1353 }
1354
1355 release_block(address, laneno)
1356 u_long address;
1357 u_char laneno;
1358 {
1359     /* This routine releases one pattern block back to the correct free list.
1360     * "address" can point anywhere in the block.
1361     */
1362
1363     u_long link_table_addr;
1364     u_long block_addr;
1365     u_long temp;
1366     u_long temp2;
1367
1368     if (address == NULL)
1369         return;
1370
1371     address -= BLOCK_START_MASK;
1372
1373     DPRINTF(("Release block addr: %08X\n", address));
1374
1375     link_table_addr = ptol(address);
1376
1377     /* Mark this block free. */
1378     write_loc_long((u_long *)link_table_addr, (u_long)FREE_BLOCK_FLAG);
1379
1380     temp = link_table_addr - LINK_TABLE_ADDR_INC;
1381     temp2 = read_loc_long((u_long *)temp) & BLOCK_NUMBER_MASK;
1382     while (temp2 != FREE_BLOCK_FLAG) {
1383
1384         temp = LINK_TABLE_ADDR_INC;
1385         temp2 = read_loc_long((u_long *)temp) & BLOCK_NUMBER_MASK;
1386     }
1387
1388     if (temp == ((address & LANE_ADDR_MASK) + LANE_LINK_TABLE_OFFSET)) {
1389         /* TEMP is pointing to the link table location corresponding to block
1390         * number 0. So none of the blocks before this block is free.
1391         * Insert it at the head of the free list.
1392         */
1393
1394         if (free_block_list(laneno) != NULL) {
1395             write_loc_long((u_long *)free_block_list(laneno) + 4, address);
1396         }
1397         write_loc_long((u_long *)address, free_block_list(laneno));
1398         write_loc_long((u_long *)address + 4, (u_long)NULL);
1399         free_block_list(laneno) = address;
1400     }
1401     else {
1402         /* Insert the block after temp */
1403
1404         block_addr = (temp & LANE_ADDR_MASK) +
1405             (temp << (BLOCK_NUMBER_SHIFT - 2));
1406
1407         DPRINTF(("Insert block after block addr: %08X\n", block_addr));
1408
1409         temp2 = read_loc_long((u_long *)block_addr);
1410
1411         write_loc_long((u_long *)address, temp2);
1412         write_loc_long((u_long *)address + 4, block_addr);
1413
1414         write_loc_long((u_long *)block_addr, address);
1415         write_loc_long((u_long *)temp2 + 4, address);
1416     }
1417 }
1418
1419 read_branch_table_content(address)
1420 u_long address;
1421 {
1422     /* Read the content of the branch table corresponding to the block with
1423     * the given address.
1424     */
1425
1426     u_long link_table_addr;
1427     u_long value;
1428
1429     address -= BLOCK_START_MASK;
1430
1431     link_table_addr = ptol(address);
1432
1433     value = read_loc_long((u_long *)link_table_addr) & BLOCK_NUMBER_MASK;
1434
1435     return(value);
1436 }
1437
1438
1439
1440

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 13/163
LINE #		SOURCE TEXT		
1441				
1442				
1443		write_patterns(instance, unit_addr, ptrs_ptr)		
1444		INSTANCE_INFO *instance;		
1445		u_long unit_addr;		
1446		PTRN_BITS_Lane_OSD *ptrs_ptr;		
1447		{		
1448		/* Write the patterns pointed to by *ptrs_ptr to address specified in		
1449		*unit_addr array.		
1450		*/		
1451		DAB_INFO *dab_ptr;		
1452		u_long addr;		
1453		u_char unitno;		
1454		u_char laneso;		
1455		u_char total_unit;		
1456				
1457		dab_ptr = dab_list(instance->dab_info_index);		
1458		total_unit = dab_ptr->unit_count;		
1459				
1460		#ifdef DEBUG		
1461		if (unit_addr[0] == instance->lcycdb_addr[0]) {		
1462		DPRINTF(("LCYCDBS :"));		
1463		}		
1464		else if (unit_addr[0] == instance->lcycmdb_addr[0]) {		
1465		DPRINTF(("LCYCDBS :"));		
1466		}		
1467		else {		
1468		DPRINTF(("PTRN 408X:", unit_addr[0]));		
1469		}		
1470				
1471		for (unitno = 0; unitno < total_unit; ++unitno) {		
1472		DPRINTF((" 408X 408X 408X", ptrs_ptr[unitno].word[0],		
1473		ptrs_ptr[unitno].word[1],		
1474		ptrs_ptr[unitno].word[2]));		
1475		}		
1476		#endif		
1477				
1478		for (unitno = 0; unitno < total_unit; ++unitno) {		
1479		addr = unit_addr[unitno];		
1480		write_loc_long((u_long *)addr,		
1481		ptrs_ptr[unitno].word[0]);		
1482		write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET),		
1483		ptrs_ptr[unitno].word[1]);		
1484		write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET),		
1485		ptrs_ptr[unitno].word[2]);		
1486		}		
1487				
1488		total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;		
1489		for (unitno = 0; unitno < total_unit; ++unitno) {		
1490		addr = unit_addr[unitno];		
1491		laneso = dab_ptr->unit_location[unitno].lane_no;		
1492				
1493		write_loc_long((u_long *)addr, dab_ptr->dummy_ptr[laneso].word[0]);		
1494		write_loc_long((u_long *)(&addr + LANE_SEGMENT_B_OFFSET),		
1495		dab_ptr->dummy_ptr[laneso].word[1]);		
1496		write_loc_long((u_long *)(&addr + LANE_SEGMENT_C_OFFSET),		
1497		dab_ptr->dummy_ptr[laneso].word[2]);		
1498		}		
1499				
1500				
1501				
1502		grow_patterns(instance)		
1503		INSTANCE_INFO *instance;		
1504		{		
1505		DAB_INFO *dab_ptr;		
1506		LANE_ADDR_INFO *lane_info;		
1507		u_long cur_unit_addr;		
1508		u_long new_unit_addr;		
1509		addr_inc_per_lane;		
1510		u_char laneso;		
1511		u_char unitno;		
1512		u_char total_unit;		
1513		u_char junk;		
1514				
1515		dab_ptr = dab_list(instance->dab_info_index);		
1516				
1517		instance->pattern_count += dab_ptr->unit_count_per_lane;		
1518				
1519		addr_inc_per_lane = dab_ptr->unit_count_per_lane * PTRN_ADDR_INC;		
1520				
1521		total_unit = dab_ptr->lane_count * dab_ptr->unit_count_per_lane;		
1522				
1523		cur_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0];		
1524		instance->cur_unit_addr_index = instance->cur_unit_addr_index + 1 & 1;		
1525		new_unit_addr = instance->unit_addr[instance->cur_unit_addr_index][0];		
1526		for (unitno = 0; unitno < total_unit; ++unitno) {		
1527		lane_info = instance->lane_addr[dab_ptr->unit_location[unitno].lane_no];		
1528		if ((cur_unit_addr[unitno] + addr_inc_per_lane) >=		
1529		lane_info->max_addr) {		
1530		if (lane_info->new_block_addr == 0) {		
1531		if (new_block(lane_info->new_block_addr,		
1532		junk,		
1533		PATTERN_BLOCK,		
1534		dab_ptr->unit_location[unitno].lane_no) == FAILURE) {		
1535		lane_info->new_block_addr = 0; /* out of Fast Pattern Memory */;		
1536		return(FAILURE);		
1537		return(FAILURE);		
1538		}		
1539				
1540		set_branch(lane_info->max_addr - PTRN_ADDR_INC,		
1541		lane_info->new_block_addr);		
1542				
1543				
1544		new_unit_addr[unitno] = lane_info->new_block_addr +		
1545		cur_unit_addr[unitno] + addr_inc_per_lane -		
1546		lane_info->max_addr;		
1547				
1548		}		
1549		else {		
1550		new_unit_addr[unitno] = cur_unit_addr[unitno] + addr_inc_per_lane;		
1551		}		
1552				
1553		if (dab_ptr->unit_location[unitno].last_in_lane)		
1554		lane_info->last_unit_addr = new_unit_addr[unitno];		
1555		}		
1556				
1557		for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {		
1558		lane_info = instance->lane_addr[laneso];		
1559				
1560		if (lane_info->new_block_addr != 0) {		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89 PAGE #  
TIME 6:14:53 pm 14/164

```

1561     lane_info->prev_max_addr = lane_info->max_addr;
1562     lane_info->new_addr = lane_info->new_block_addr + BLOCK_ADDR_INC;
1563     lane_info->new_block_addr = 0;
1564 }
1565 }
1566 }
1567 return(SUCCESS);
1568 }
1569
1570 set_branch(cur_ptrn_addr, current_block_addr)
1571 u_long cur_ptrn_addr;
1572 u_long current_block_addr;
1573 {
1574     /* Set the branch command BRANCH_LATENCY ptrns before the current pattern.
1575     * Modify the branch table appropriately.
1576     */
1577     u_long temp;
1578     u_long branch_addr;
1579
1580     branch_addr = cur_ptrn_addr - BRANCH_LATENCY * PTRN_ADDR_INC +
1581                 LANE_SEGMENT_C_OFFSET;
1582     temp = read_loc_long((u_long *)branch_addr);
1583     temp = (temp & PEL_BRANCH_MASK) | (BRANCH_ALWAYS << PEL_BRANCH_SHIFT);
1584     write_loc_long((u_long *)branch_addr, temp);
1585     write_branch_table(cur_ptrn_addr, current_block_addr);
1586 }
1587
1588 unset_branch(cur_ptrn_addr)
1589 u_long cur_ptrn_addr;
1590 {
1591     /* Unset the branch command BRANCH_LATENCY ptrns before the current pattern.
1592     * Modify the branch table appropriately.
1593     */
1594     u_long temp;
1595     u_long branch_addr;
1596
1597     branch_addr = cur_ptrn_addr - BRANCH_LATENCY * PTRN_ADDR_INC +
1598                 LANE_SEGMENT_C_OFFSET;
1599     temp = read_loc_long((u_long *)branch_addr);
1600     temp = (temp & PEL_BRANCH_MASK) | (BRANCH_NEVER << PEL_BRANCH_SHIFT);
1601     write_loc_long((u_long *)branch_addr, temp);
1602     /* Write 0 to the link table entry corresponding to cur_ptrn_addr
1603     * to indicate that this is the last block of the sequence.
1604     */
1605     write_branch_table(cur_ptrn_addr, (u_long)0);
1606 }
1607
1608 write_branch_table(source, destination)
1609 u_long source;
1610 u_long destination;
1611 {
1612     /* This routine writes the "destination" address into the branch table
1613     * location corresponding to the pattern block containing address
1614     * "source". i.e. "source" can point to be anywhere in the block.
1615     * Note that the contents of the link table is the "Block Number" of
1616     * the next block to jump to (not the actual address of the block).
1617     */
1618     u_long link_table_addr;
1619     source &= BLOCK_START_MASK;
1620     link_table_addr = ptol(source);
1621     write_loc_long((u_long *)link_table_addr,
1622                 (destination >> BLOCK_NUMBER_SHIFT) & BLOCK_NUMBER_MASK);
1623 }
1624
1625 grow_pattern_unit(instance)
1626 INSTANCE_INFO *instance;
1627 {
1628     /* This routine grows the last_in_lane unit pattern address on each lane
1629     * by just one unit.
1630     * It is assumed that there are enough rooms in the block for the pattern
1631     * to grow; it will not allocate new pattern block.
1632     */
1633     DAB_INFO *dab_ptr;
1634     LANE_ADDR_INFO *lane_info;
1635     u_char lanes;
1636
1637     dab_ptr = dab_list(instance->dab_info_index);
1638     instance->pattern_count += 1;
1639     for (lanes = 0; lanes < MAX_LANE_COUNT; ++lanes) {
1640         lane_info = &instance->lane_addr[lanes];
1641         if (dab_ptr->lane_used[lanes]) {
1642             if ((lane_info->last_unit_addr + PTRN_ADDR_INC >=
1643                 lane_info->max_addr) {
1644                 lm_queue_message(ERROR_MSG, "Internal error: grow_pattern_unit pattern address overflow");
1645                 return(FAILURE);
1646             }
1647             lane_info->last_unit_addr += PTRN_ADDR_INC;
1648         }
1649     }
1650     return(SUCCESS);
1651 }
1652
1653 adjust_pattern_addr(instance)
1654 INSTANCE_INFO *instance;
1655 {
1656     /* This routine adjust the LANE_ADDR_INFO.last_unit_addr to point to
1657     * the address just past the last full lane pattern address.
1658     * It is assumed that subtracting the address does not make the address
1659     * go outside the block address boundary.
1660     */
1661     DAB_INFO *dab_ptr;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 15/165
LINE #		SOURCE TEXT		
1681		u_char laneso;		
1682		dab_ptr = dab_list(instance->dab_info_index);		
1683		for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {		
1684		if (dab_ptr->lane_used[laneso]) {		
1685		instance->lane_addr[laneso].last_unit_addr --		
1686		(dab_ptr->unit_count_per_lane - 1) * PTRN_ADDR_INC;		
1687		}		
1688		instance->pattern_count = instance->pattern_count -		
1689		dab_ptr->unit_count_per_lane + 1;		
1690		}		
1691		write_patterns_unit(instance, ptrn_ptr)		
1692		INSTANCE_INFO *instance;		
1693		PTRN_BITS_LONGWORD *ptrn_ptr;		
1694		{		
1695		/* Use the addresses for the last in lane unit to write the 1 unit worth		
1696		* of patterns pointed by ptrn_ptr.		
1697		*/		
1698		DAB_INFO *dab_ptr;		
1699		LANE_ADDR_INFO *lane_info;		
1700		u_long addr;		
1701		u_char laneso;		
1702		dab_ptr = dab_list(instance->dab_info_index);		
1703		{		
1704		for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {		
1705		lane_info = instance->lane_addr[laneso];		
1706		if (dab_ptr->lane_used[laneso]) {		
1707		addr = lane_info->last_unit_addr;		
1708		write_loc_long((u_long *)addr,		
1709		ptrn_ptr[laneso].word(0));		
1710		write_loc_long((u_long *) (addr + LANE_SEGMENT_B_OFFSET),		
1711		ptrn_ptr[laneso].word(1));		
1712		write_loc_long((u_long *) (addr + LANE_SEGMENT_C_OFFSET),		
1713		ptrn_ptr[laneso].word(2));		
1714		}		
1715		#ifdef DEBUG		
1716		DPRINTF(("PTRN UNIT %08X: %08X %08X %08X\n", addr,		
1717		ptrn_ptr[laneso].word(0),		
1718		ptrn_ptr[laneso].word(1),		
1719		ptrn_ptr[laneso].word(2));		
1720		#endif		
1721		}		
1722		}		
1723		copy_patterns(source_addr, dest_addr, count)		
1724		u_long source_addr;		
1725		u_long dest_addr;		
1726		u_short count;		
1727		{		
1728		/* This routine copy "count" number of unit patterns from "source_addr"		
1729		* to "dest_addr".		
1730		*/		
1731		u_long cur_source_addr;		
1732		u_long cur_dest_addr;		
1733		u_long value;		
1734		u_short i;		
1735		DPRINTF(("copy_patterns: source_addr: %08X, dest_addr: %08X\n",		
1736		source_addr, dest_addr));		
1737		cur_source_addr = source_addr;		
1738		cur_dest_addr = dest_addr;		
1739		for (i = 0; i < count; ++i) {		
1740		value = read_loc_long((u_long *)cur_source_addr);		
1741		write_loc_long((u_long *)cur_dest_addr, value);		
1742		value =		
1743		read_loc_long((u_long *) (cur_source_addr + LANE_SEGMENT_B_OFFSET));		
1744		write_loc_long((u_long *) (cur_dest_addr + LANE_SEGMENT_B_OFFSET), value);		
1745		value =		
1746		read_loc_long((u_long *) (cur_source_addr + LANE_SEGMENT_C_OFFSET));		
1747		write_loc_long((u_long *) (cur_dest_addr + LANE_SEGMENT_C_OFFSET), value);		
1748		cur_source_addr += PTRN_ADDR_INC;		
1749		cur_dest_addr += PTRN_ADDR_INC;		
1750		}		
1751		}		
1752		extend_inst_table(user)		
1753		USER_INFO *user;		
1754		{		
1755		INSTANCE_INFO **temp_ptr;		
1756		char *ptr;		
1757		long new_size;		
1758		u_long i;		
1759		new_size = user->inst_table_size + INST_TABLE_INCR;		
1760		ptr = (char *)		
1761		DREALLOC((char *)user->instance,		
1762		(unsigned)(new_size * sizeof(INSTANCE_INFO)));		
1763		if (ptr == NULL)		
1764		return(FAILURE);		
1765		user->instance = (INSTANCE_INFO **)ptr;		
1766		user->free_inst_id = (INSTANCE_INFO *)		
1767		user->instance[user->inst_table_size];		
1768		temp_ptr = user->instance[user->inst_table_size + 1];		
1769		for (i = user->inst_table_size; i < (new_size - 1); i++) {		
1770		{		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

16/166

LINE # SOURCE TEXT

```

1801     user->instance[i] = (INSTANCE_INFO *)temp_ptr;
1802     temp_ptr++;
1803 }
1804
1805     user->instance[i] = NULL;
1806     user->inst_table_size = new_size;
1807     return(SUCCESS);
1808 }
1809
1810
1811 extend_def_table(user)
1812 USER_INFO *user;
1813 {
1814     DEVICE_SPEC **temp_ptr;
1815     char *ptr;
1816     long new_size;
1817     u_long i;
1818
1819     new_size = user->def_table_size + DEF_TABLE_INCR;
1820
1821     ptr = (char *)
1822     DREALLOC((char *)user->definition,
1823             (unsigned)(new_size * sizeof(DEVICE_SPEC *)));
1824     if (ptr == NULL)
1825         return(FAILURE);
1826
1827     user->definition = (DEVICE_SPEC **)ptr;
1828
1829     user->free_def_id = user->definition[user->def_table_size];
1830
1831     temp_ptr = (DEVICE_SPEC **)
1832     user->definition[user->def_table_size + 1];
1833
1834     for (i = user->def_table_size; i < (new_size - 1); i++) {
1835         user->definition[i] = (DEVICE_SPEC *)temp_ptr;
1836         ++temp_ptr;
1837     }
1838
1839     user->definition[i] = NULL;
1840     user->def_table_size = new_size;
1841
1842     return(SUCCESS);
1843 }
1844
1845 new_and_link_instance(user, def_ptr, name_ptr,
1846                      unit_count, inst_id_table_index,
1847                      dab_info_index)
1848 USER_INFO *user;
1849 DEVICE_SPEC *def_ptr;
1850 char *name_ptr;
1851 u_short unit_count;
1852 u_short inst_id_table_index;
1853 char *dab_info_index;
1854 {
1855     INSTANCE_INFO *temp_ptr;
1856     INSTANCE_INFO *inst_ptr;
1857     PTRN_BITS *ptrn_bits_ptr;
1858     PIN_INFO *pin;
1859     DAB_INFO *dab_ptr;
1860     EXTRA_DEVICE_SPEC *extra_def_ptr;
1861     PTRN_BITS_LONGWORD *unit_ptr;
1862     PIN_SPEC *pin_spec_ptr;
1863     long pinno;
1864     u_char unitno;
1865     u_char wordno;
1866     u_char alloc_error = FALSE;
1867
1868     temp_ptr = (INSTANCE_INFO *)DCALLOC((unsigned)1,
1869                                         (unsigned)sizeof(INSTANCE_INFO));
1870     if (temp_ptr == NULL)
1871         return(FAILURE);
1872
1873     temp_ptr->device_info_string = (char *)
1874     DREALLOC((unsigned)(strlen(name_ptr) + 1));
1875
1876     if (temp_ptr->device_info_string == NULL) {
1877         DFREE((char *)temp_ptr);
1878         return(FAILURE);
1879     }
1880
1881     (void)strcpy(temp_ptr->device_info_string, name_ptr);
1882
1883     temp_ptr->definition = def_ptr;
1884     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
1885
1886     temp_ptr->dab_info_index = dab_info_index;
1887     temp_ptr->failed_to_alloc_ptrn = FALSE;
1888     temp_ptr->fatal_error = FALSE;
1889     temp_ptr->had_aborted_pin = FALSE;
1890
1891     temp_ptr->restore_state = SENT_RECV_NOTHING;
1892
1893     dab_ptr = dab_list[temp_ptr->dab_info_index];
1894
1895     temp_ptr->pin_info_table =
1896     (PIN_INFO *)DREALLOC((unsigned)(def_ptr->pin_cat * sizeof(PIN_INFO)));
1897     if (temp_ptr->pin_info_table == NULL)
1898         alloc_error = TRUE;
1899     else {
1900         /* Firstly, clear the entire pin_info_table. */
1901         bzero((char *)temp_ptr->pin_info_table,
1902              (int)(def_ptr->pin_cat * sizeof(PIN_INFO)));
1903         /* Secondly, initialize each element of the pin_info_table */
1904         for (pinno = def_ptr->pin_cat - 1; pinno >= 0; --pinno) {
1905             pin_spec_ptr = extra_def_ptr->pin_table[pinno];
1906             pin = temp_ptr->pin_info_table[pinno];
1907
1908             pin->old_raw = LOGIC_U;
1909             pin->old_filtered = LOGIC_U;
1910             pin->new_raw = LOGIC_U;
1911             pin->new_filtered = LOGIC_U;
1912             pin->mis_delay = -1;
1913             pin->sim_time = 0;
1914             pin->next_input_pin_index = -1;
1915             pin->next_output_pin_index = -1;
1916             pin->input_pin_is_linked = FALSE;
1917             pin->output_pin_is_linked = FALSE;
1918             pin->delay_type = DELAY_TABLE;
1919             pin->uninitialized_pin = TRUE;
1920             pin->has_resistor = NO_PULLUP_OR_PULLDOWN;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 17/167
LINE #	SOURCE TEXT			
1921	pin->direction = pin_spec_ptr->direction;			
1922				
1923	if ((pin_spec_ptr->direction == IN)			
1924	(pin_spec_ptr->direction == OUT)			
1925	(pin_spec_ptr->direction == IO)) {			
1926				
1927	if (pin_spec_ptr->pulled_up == 1) {			
1928	pin->has_resistor = PULLUP;			
1929	}			
1930				
1931	if (pin_spec_ptr->pulled_down == 1) {			
1932	pin->has_resistor = PULLDOWN;			
1933	}			
1934	}			
1935				
1936				
1937				
1938	temp_ptr->first_data_pin_index = -1;			
1939	temp_ptr->first_eval_pin_index = -1;			
1940	temp_ptr->first_store_pin_index = -1;			
1941				
1942	temp_ptr->is_fault = FALSE;			
1943	temp_ptr->enable_timing_meas = FALSE;			
1944	temp_ptr->use_2_bit_per_pin = FALSE;			
1945	temp_ptr->fault_count = 0;			
1946	temp_ptr->check_input_count = MAX_CHECK_INPUT_COUNT;			
1947	temp_ptr->sample_count = DEFAULT_SAMPLE_COUNT;			
1948	temp_ptr->evaluation_count = DEFAULT_EVALUATION_COUNT;			
1949				
1950	temp_ptr->first_eval = TRUE;			
1951				
1952	temp_ptr->ptrs_loaded =			
1953	(PTRN_BITS *)DCALLOC((unsigned)unit_count,			
1954	(PTRN_BITS *)DCALLOC((unsigned)sizeof(PTRN_BITS)));			
1955	if (temp_ptr->ptrs_loaded == NULL)			
1956	alloc_error = TRUE;			
1957	else {			
1958	/* Initialize the ptrs_loaded to all 1's */			
1959	for (unitno = 0; unitno < unit_count; ++unitno) {			
1960	unit_ptr = (PTRN_BITS *)DCALLOC((unsigned)sizeof(PTRN_BITS),			
1961	unit_ptr->word[0] = 0xffffffff;			
1962	unit_ptr->word[1] = 0xffffffff;			
1963	unit_ptr->word[2] = 0xffffffff;			
1964	}			
1965				
1966	/* Setup the pattern control bits */			
1967	ptrs_bits_ptr = temp_ptr->ptrs_loaded;			
1968	for (unitno = 0; unitno < unit_count; ++unitno) {			
1969				
1970	if (dab_ptr->unit_location[unitno].last_in_lane)			
1971	set_pel_ctl(ptrs_bits_ptr->ctl, PEL_CTL_LOAD_DATA_LAST_UNIT);			
1972	else			
1973	set_pel_ctl(ptrs_bits_ptr->ctl, PEL_CTL_LOAD_DATA);			
1974				
1975	set_pel_sel(ptrs_bits_ptr->ctl,			
1976	dab_ptr->unit_location[unitno].slot_no);			
1977				
1978	/* Disable the R1 clocks.			
1979	/* Note that the R1 clocks are already disabled because we initialize			
1980	ptrs_loaded to 0.			
1981	*/			
1982	for (wordno = 0; wordno < 5; ++wordno) {			
1983	ptrs_bits_ptr->word[wordno]  =			
1984	((PTRN_BITS *)DCALLOC((unsigned)sizeof(PTRN_BITS))->word[wordno]);			
1985	}			
1986				
1987	++ptrs_bits_ptr;			
1988				
1989	}			
1990				
1991	/* HWDEN_LOADED, LITCHES_LOADED, LATCHES_LOADED are initialized in			
1992	load_preamble().			
1993	*/			
1994	temp_ptr->hwdenb_loaded =			
1995	(PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS));			
1996	if (temp_ptr->hwdenb_loaded == NULL)			
1997	alloc_error = TRUE;			
1998				
1999	temp_ptr->leychdb_loaded =			
2000	(PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS));			
2001	if (temp_ptr->leychdb_loaded == NULL)			
2002	alloc_error = TRUE;			
2003				
2004	temp_ptr->leymdb_loaded =			
2005	(PTRN_BITS *)DCALLOC((unsigned)unit_count * sizeof(PTRN_BITS));			
2006	if (temp_ptr->leymdb_loaded == NULL)			
2007	alloc_error = TRUE;			
2008				
2009	temp_ptr->last_consistent_set =			
2010	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2011	if (temp_ptr->last_consistent_set == NULL)			
2012	alloc_error = TRUE;			
2013				
2014	temp_ptr->sim_pin_value.data =			
2015	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2016	if (temp_ptr->sim_pin_value.data == NULL)			
2017	alloc_error = TRUE;			
2018				
2019	temp_ptr->sim_pin_value.hiz =			
2020	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2021	if (temp_ptr->sim_pin_value.hiz == NULL)			
2022	alloc_error = TRUE;			
2023				
2024	temp_ptr->sim_pin_value.unknown =			
2025	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2026	if (temp_ptr->sim_pin_value.unknown == NULL)			
2027	alloc_error = TRUE;			
2028				
2029	temp_ptr->sim_pin_value.soft =			
2030	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2031	if (temp_ptr->sim_pin_value.soft == NULL)			
2032	alloc_error = TRUE;			
2033				
2034	temp_ptr->last_sample_value.data =			
2035	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			
2036	if (temp_ptr->last_sample_value.data == NULL)			
2037	alloc_error = TRUE;			
2038				
2039	temp_ptr->last_sample_value.hiz =			
2040	(PTRN_BITS *)DCALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS));			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
18/168

```

LINE # SOURCE TEXT
2041 if (temp_ptr->last_sample_value.hiz == NULL)
2042     alloc_error = TRUE;
2043
2044 temp_ptr->last_sample_value.unknown =
2045 (PTRN_BITS * DALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)));
2046 if (temp_ptr->last_sample_value.unknown == NULL)
2047     alloc_error = TRUE;
2048
2049 temp_ptr->last_sample_value.soft =
2050 (PTRN_BITS * DALLOC((unsigned)unit_count, (unsigned)sizeof(PTRN_BITS)));
2051 if (temp_ptr->last_sample_value.soft == NULL)
2052     alloc_error = TRUE;
2053
2054 if (user->free_inst_id == NULL)
2055     if (extend_inst_table(user) == FAILURE)
2056         alloc_error = TRUE;
2057
2058 if (alloc_error == TRUE) {
2059     DFREE((char *)temp_ptr->device_info_string);
2060     DFREE((char *)temp_ptr->pin_info_table);
2061     DFREE((char *)temp_ptr->last_consistent_set);
2062     DFREE((char *)temp_ptr->ptrn_loaded);
2063     DFREE((char *)temp_ptr->lcyddb_loaded);
2064     DFREE((char *)temp_ptr->lcyddb_loaded);
2065     DFREE((char *)temp_ptr->sim_pin_value.data);
2066     DFREE((char *)temp_ptr->sim_pin_value.unknown);
2067     DFREE((char *)temp_ptr->sim_pin_value.soft);
2068     DFREE((char *)temp_ptr->last_sample_value.data);
2069     DFREE((char *)temp_ptr->last_sample_value.hiz);
2070     DFREE((char *)temp_ptr->last_sample_value.unknown);
2071     DFREE((char *)temp_ptr->last_sample_value.soft);
2072     DFREE((char *)temp_ptr);
2073     return(FAILURE);
2074 }
2075
2076 slot_ptr = user->free_inst_id;
2077
2078 *inst_id_table_index =
2079 (u_short)((u_long)slot_ptr - (u_long)user->instance) /
2080     sizeof(INSTANCE_INFO *);
2081
2082 user->free_inst_id = (INSTANCE_INFO *)slot_ptr;
2083 *slot_ptr = temp_ptr;
2084
2085 return(SUCCESS);
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100 PTRN BITS      *ptrn_bits_ptr;
2101 PIN INFO      *pin;
2102 DAB INFO      *dab_ptr;
2103 DEVICE SPEC   *def_ptr;
2104 EXTRA_DEVICE SPEC *extra_def_ptr;
2105 PTRN BITS LONGWORD *unit_ptr;
2106 PTRN BITS LONGWORD *last_consistent_set_ptr;
2107 PTRN BITS LONGWORD *sim_pin_value_data_ptr;
2108 PTRN BITS LONGWORD *sim_pin_value_hiz_ptr;
2109 PTRN BITS LONGWORD *sim_pin_value_unknown_ptr;
2110 PTRN BITS LONGWORD *sim_pin_value_soft_ptr;
2111 PTRN BITS LONGWORD *last_sample_value_data_ptr;
2112 PTRN BITS LONGWORD *last_sample_value_hiz_ptr;
2113 PTRN BITS LONGWORD *last_sample_value_unknown_ptr;
2114 PTRN BITS LONGWORD *last_sample_value_soft_ptr;
2115 long          pinno;
2116 u_char        unit_count;
2117 u_char        unitno;
2118 u_char        wordno;
2119 u_char        lanes;
2120
2121 def_ptr = instance->definition;
2122 extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
2123 dab_ptr = dab_list(instance->dab_info_index);
2124 unit_count = dab_ptr->unit_count;
2125
2126 instance->pattern_count = 0;
2127 instance->static_pattern_count = 0;
2128 instance->common_pattern_count = 0;
2129
2130 for (laneno = 0; laneno < MAX_LANE_COUNT; ++laneno) {
2131     instance->fb_block_size[laneno] = 0;
2132     instance->fb_block_addr[laneno] = 0;
2133     instance->lane_addr[laneno].max_addr = 0;
2134     instance->lane_addr[laneno].prev_max_addr = 0;
2135     instance->lane_addr[laneno].last_unit_addr = 0;
2136     instance->lane_addr[laneno].new_block_addr = 0;
2137     instance->seq_start_addr[laneno] = 0;
2138     instance->var_seq_addr[laneno] = 0;
2139     instance->last_common_block_addr[laneno] = 0;
2140 }
2141
2142 for (unitno = 0; unitno < MAX_UNIT_COUNT; ++unitno) {
2143     instance->unit_addr[0][unitno] = 0;
2144     instance->unit_addr[1][unitno] = 0;
2145     instance->first_user_ptrn_unit_addr[unitno] = 0;
2146     instance->lcyddb_addr[unitno] = 0;
2147     instance->lcyddb_addr[unitno] = 0;
2148 }
2149
2150 instance->failed_to_alloc_ptrn = FALSE;
2151 instance->restore_state = SENT_RECV_NOTHING;
2152
2153 instance->cur_unit_addr_index = 0;
2154 instance->has_history = FALSE;
2155 instance->has_shorted_pin = FALSE;
2156 instance->purge_ptrn_on_next_eval = FALSE;
2157
2158 if (def_ptr->device_type == PRIVATE) {
2159     instance->has_history = TRUE;
2160     instance->purge_ptrn_on_next_eval = TRUE;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	19/169

```

2161
2162
2163 /* Initialize pin info table */
2164 for (pinno = def_ptr->pin_cat - 1; pinno >= 0; --pinno) {
2165     pin = instance->pin_info_table[pinno];
2166     pin->old_raw = LOGIC_U;
2167     pin->old_filtered = LOGIC_U;
2168     pin->new_raw = LOGIC_U;
2169     pin->new_filtered = LOGIC_U;
2170     pin->min_delay = -1;
2171     pin->min_time = 0;
2172     pin->next_input_pin_index = -1;
2173     pin->next_output_pin_index = -1;
2174     pin->input_pin_is_linked = FALSE;
2175     pin->output_pin_is_linked = FALSE;
2176     pin->delay_type = DELAY_TABLE;
2177     pin->uninitialized_pin = TRUE;
2178 }
2179 instance->first_data_pin_index = -1;
2180 instance->first_eval_pin_index = -1;
2181 instance->first_start_pin_index = -1;
2182
2183 instance->is_fault = FALSE;
2184 instance->enable_timing_warnings = FALSE;
2185 instance->use_2_bit_per_pin = FALSE;
2186 instance->fault_count = 0;
2187 instance->check_input_z_count = MAX_CHECK_INPUT_Z_COUNT;
2188 instance->sample_count = DEFAULT_SAMPLE_COUNT;
2189 instance->evaluation_count = DEFAULT_EVALUATION_COUNT;
2190
2191 instance->first_eval = TRUE;
2192
2193 /* Initialize the ptrs loaded to all 1 */
2194 for (unitno = 0; unitno < unit_count; ++unitno) {
2195     unit_ptrs = (PTRN_BITS_LONGWORD *)instance->ptrs_loaded[unitno];
2196     unit_ptrs->word[0] = 0xffffffff;
2197     unit_ptrs->word[1] = 0xffffffff;
2198     unit_ptrs->word[2] = 0xffffffff;
2199 }
2200
2201 /* Setup the pathbars control bits */
2202 ptrs_bits_ptr = instance->ptrs_loaded;
2203 for (unitno = 0; unitno < unit_count; ++unitno) {
2204     if (dab_ptr->unit_location[unitno].last_in_lane)
2205         set_ptr_ctl(ptrs_bits_ptr->ctl, PBL_CTL_LOAD_DATA_LAST_UNIT);
2206     else
2207         set_ptr_ctl(ptrs_bits_ptr->ctl, PBL_CTL_LOAD_DATA);
2208
2209     set_ptr_sel(ptrs_bits_ptr->ctl,
2210         dab_ptr->unit_location[unitno].slot_no);
2211
2212     /* Disable the R1 clocks.
2213      * Note that the R1 clocks are already disabled because we initialize
2214      * ptrs_loaded to 0.
2215      */
2216     for (wordno = 0; wordno < 5; ++wordno) {
2217         ptrs_bits_ptr->word[wordno] |=
2218             ((PTRN_BITS *)(&extra_def_ptr->ident_R1[unitno]))->word[wordno];
2219     }
2220     ++ptrs_bits_ptr;
2221 }
2222
2223 /* BUSES_LOADED, LATCHES_LOADED, LATCHES_LOADED are initialized in
2224  * load_preamble().
2225  */
2226
2227 /* Initialize the following PTRN_BITS to 0 */
2228 last_consistent_set_ptr = (PTRN_BITS_LONGWORD *)
2229     instance->last_consistent_set;
2230 sim_pin_value_data_ptr = (PTRN_BITS_LONGWORD *)
2231     instance->sim_pin_value.data;
2232 sim_pin_value_hiz_ptr = (PTRN_BITS_LONGWORD *)
2233     instance->sim_pin_value.hiz;
2234 sim_pin_value_unknown_ptr = (PTRN_BITS_LONGWORD *)
2235     instance->sim_pin_value.unknown;
2236 sim_pin_value_soft_ptr = (PTRN_BITS_LONGWORD *)
2237     instance->sim_pin_value.soft;
2238 last_sample_value_data_ptr = (PTRN_BITS_LONGWORD *)
2239     instance->last_sample_value.data;
2240 last_sample_value_hiz_ptr = (PTRN_BITS_LONGWORD *)
2241     instance->last_sample_value.hiz;
2242 last_sample_value_unknown_ptr = (PTRN_BITS_LONGWORD *)
2243     instance->last_sample_value.unknown;
2244 last_sample_value_soft_ptr = (PTRN_BITS_LONGWORD *)
2245     instance->last_sample_value.soft;
2246
2247 for (unitno = 0; unitno < unit_count; ++unitno) {
2248     for (wordno = 0; wordno < 3; ++wordno) {
2249         last_consistent_set_ptr->word[wordno] = 0;
2250
2251         sim_pin_value_data_ptr->word[wordno] = 0;
2252         sim_pin_value_hiz_ptr->word[wordno] = 0;
2253         sim_pin_value_unknown_ptr->word[wordno] = 0;
2254         sim_pin_value_soft_ptr->word[wordno] = 0;
2255
2256         last_sample_value_data_ptr->word[wordno] = 0;
2257         last_sample_value_hiz_ptr->word[wordno] = 0;
2258         last_sample_value_unknown_ptr->word[wordno] = 0;
2259         last_sample_value_soft_ptr->word[wordno] = 0;
2260     }
2261 }
2262
2263 ++last_consistent_set_ptr;
2264 ++sim_pin_value_data_ptr;
2265 ++sim_pin_value_hiz_ptr;
2266 ++sim_pin_value_unknown_ptr;
2267 ++sim_pin_value_soft_ptr;
2268
2269 ++last_sample_value_data_ptr;
2270 ++last_sample_value_hiz_ptr;
2271 ++last_sample_value_unknown_ptr;
2272 ++last_sample_value_soft_ptr;
2273
2274
2275
2276
2277
2278 rls_instance(user, inst_id_table_index)
2279 USER_INFO user;
2280 u_short inst_id_table_index;

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	20/170

```

2281 |
2282 |
2283 |     INSTANCE_INFO    *instance;
2284 |
2285 |     instance = user->instance["-> id_table_index];
2286 |
2287 |     DFREE((char *)instance->device_info_string);
2288 |
2289 |     DFREE((char *)instance->pin_info_table);
2290 |
2291 |     DFREE((char *)instance->last_consistent_set);
2292 |
2293 |     DFREE((char *)instance->ptrs_loaded);
2294 |     DFREE((char *)instance->main_pis_value.loaded);
2295 |     DFREE((char *)instance->keychdb_loaded);
2296 |     DFREE((char *)instance->keychdb_loaded);
2297 |
2298 |     DFREE((char *)instance->ain_pis_value.data);
2299 |     DFREE((char *)instance->ain_pis_value.hiz);
2300 |     DFREE((char *)instance->ain_pis_value.unknown);
2301 |     DFREE((char *)instance->ain_pis_value.soft);
2302 |
2303 |     DFREE((char *)instance->last_sample_value.data);
2304 |     DFREE((char *)instance->last_sample_value.hiz);
2305 |     DFREE((char *)instance->last_sample_value.unknown);
2306 |     DFREE((char *)instance->last_sample_value.soft);
2307 |
2308 |     DFREE((char *)instance);
2309 |
2310 |     user->instance[inst_id_table_index] = (INSTANCE_INFO *)user->free_inst_id;
2311 |     user->free_inst_id = &user->instance[inst_id_table_index];
2312 |
2313 | }
2314 |
2315 | /* Definition of user, def_id_table_index */
2316 | USER_INFO *user;
2317 | u_short   def_id_table_index;
2318 | {
2319 |     EXTRA_DEVICE_SPEC *extra_def_ptr;
2320 |     DEVICE_SPEC        *def_ptr;
2321 |
2322 |     def_ptr = user->definition[def_id_table_index];
2323 |
2324 |     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
2325 |     DFREE((char *)extra_def_ptr->ident_inputs);
2326 |     DFREE((char *)extra_def_ptr->ident_outputs);
2327 |     DFREE((char *)extra_def_ptr->ident_ios);
2328 |     DFREE((char *)extra_def_ptr->ident_RI);
2329 |     DFREE((char *)extra_def_ptr->ident_RI);
2330 |     DFREE((char *)extra_def_ptr->ident_store);
2331 |
2332 |     DFREE((char *)extra_def_ptr);
2333 |
2334 |     /* Free the DEVICE_SPEC structure by calling a procedure */
2335 |     free_device(def_ptr);
2336 |
2337 |     /* Link the free list */
2338 |     user->definition[def_id_table_index] = (DEVICE_SPEC *)user->free_def_id;
2339 |     user->free_def_id = &user->definition[def_id_table_index];
2340 | }
2341 |
2342 | set_seq_end_bit(instance, lane_addr_info, alloc_tmp_block, seq_end_addr,
2343 |                 inst_block_number)
2344 | {
2345 |     INSTANCE_INFO    *instance;
2346 |     LANE_ADDR_INFO   *lane_addr_info;
2347 |     u_char            alloc_tmp_block;
2348 |     u_long            seq_end_addr;
2349 |     u_long            inst_block_number;
2350 |
2351 |     /* The address of the last unit is specified in "lane_addr_info".
2352 |     * If "alloc_tmp_block" == FALSE then we should look at
2353 |     * instance->lane_addr[]_prev_max_addr if we need to set the
2354 |     * SEQ_END_BIT in the previous block, otherwise we should look at
2355 |     * instance->last_addr[]_max_addr.
2356 |     */
2357 |
2358 |     DAB_INFO *dab_ptr;
2359 |     u_long   addr;
2360 |     u_long   link_table_addr;
2361 |     u_long   temp;
2362 |     u_short   fault_block_number;
2363 |     u_short   ptrs_count;
2364 |     u_char    lane0;
2365 |     u_char    i;
2366 |
2367 |     dab_ptr = dab_list(instance->dab_info_index);
2368 |
2369 |     /* Fix the block link if it is a fault pattern sequence */
2370 |     if (instance->is_fault == TRUE) {
2371 |         if (instance->disjoint_flag == FALSE) {
2372 |             for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
2373 |                 if (!dab_ptr->lane_used[lane0])
2374 |                     continue;
2375 |
2376 |                 link_table_addr =
2377 |                     ptol(instance->last_common_block_addr[lane0]);
2378 |
2379 |                 inst_block_number[lane0] =
2380 |                     read_loc_long((u_long *)link_table_addr) & BLOCK_NUMBER_MASK;
2381 |
2382 |                 fault_block_number = ptob(instance->var_seq_addr[lane0]);
2383 |
2384 |                 if (instance->last_common_block_is_feedback) {
2385 |                     for (i = 0; i < instance->fb_block_size[lane0]; ++i) {
2386 |                         write_loc_long((u_long *)link_table_addr,
2387 |                                         (u_long)fault_block_number);
2388 |                         link_table_addr += LINK_TABLE_ADDR_INC;
2389 |                     }
2390 |                 }
2391 |                 else {
2392 |                     write_loc_long((u_long *)link_table_addr,
2393 |                                     (u_long)fault_block_number);
2394 |                 }
2395 |             }
2396 |         }
2397 |     }
2398 |
2399 |     for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
2400 |         if (dab_ptr->lane_used[lane0]) {

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 21/171
LINE #		SOURCE TEXT		
2401		addr = lane_addr[instance->lane].last_unit_addr;		
2402		ptrs_count = PTRN_COUNT_ON_BLOCK(addr);		
2403		if (ptrs_count <= SEQ_END_LATENCY) {		
2404		if (alloc_temp_block == TRUE) {		
2405		addr = instance->lane_addr[instance->lane].max_addr -		
2406		(SEQ_END_LATENCY + 1 - ptrs_count) * PTRN_ADDR_INC +		
2407		LANE_SEGMENT_C_OFFSET;		
2408		}		
2409		else {		
2410		addr = instance->lane_addr[instance->lane].prev_max_addr -		
2411		(SEQ_END_LATENCY + 1 - ptrs_count) * PTRN_ADDR_INC +		
2412		LANE_SEGMENT_C_OFFSET;		
2413		}		
2414		/* The SEQ_END bit should be set in the current block. */		
2415		addr = addr - SEQ_END_LATENCY * PTRN_ADDR_INC +		
2416		LANE_SEGMENT_C_OFFSET;		
2417		seq_end_addr[instance] = addr;		
2418		DPRINTF(("setting SEQ_END bit at addr: %08X\n", addr));		
2419		temp = read_loc_long((u_long *)addr);		
2420		write_loc_long((u_long *)addr, (u_long)(temp   ~FEL_STOP_MASK));		
2421		}		
2422		}		
2423		}		
2424		remove_seq_end_bit(instance, seq_end_addr, inst_block_number);		
2425		INSTANCE_INFO *instance;		
2426		u_long seq_end_addr;		
2427		u_long inst_block_number;		
2428		DAB_INFO *dab_ptr;		
2429		u_long temp;		
2430		u_long link_table_addr;		
2431		u_char lane;		
2432		i;		
2433		dab_ptr = dab_list[instance->dab_info_index];		
2434		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
2435		if (dab_ptr->lane_used[lane]) {		
2436		DPRINTF(("removing seq end bit addr: %08X\n", seq_end_addr[instance]));		
2437		temp = read_loc_long((u_long *)seq_end_addr[instance]);		
2438		write_loc_long((u_long *)seq_end_addr[instance],		
2439		(u_long)(temp & FEL_STOP_MASK));		
2440		}		
2441		/* Restore the block link if it is a fault pattern sequence. */		
2442		if (instance->is_fault == TRUE) {		
2443		if (instance->disjoint_flag == FALSE) {		
2444		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
2445		if (!dab_ptr->lane_used[lane])		
2446		continue;		
2447		link_table_addr =		
2448		ptol(instance->last_common_block_addr[lane]);		
2449		if (instance->last_common_block_is_feedback) {		
2450		for (i = 0; i < instance->fb_block_size[lane]; ++i) {		
2451		write_loc_long((u_long *)link_table_addr,		
2452		(u_long)inst_block_number[lane]);		
2453		link_table_addr += LINK_TABLE_ADDR_INC;		
2454		}		
2455		else {		
2456		write_loc_long((u_long *)link_table_addr,		
2457		(u_long)inst_block_number[lane]);		
2458		}		
2459		}		
2460		}		
2461		play_ptr_seq(instance, timeout, changed_dac)		
2462		INSTANCE_INFO *instance;		
2463		u_long timeout;		
2464		u_char changed_dac;		
2465		{		
2466		EXTRA_DEVICE_SPEC *extra_def_ptr;		
2467		u_long start_time;		
2468		u_long elapsed;		
2469		u_char replay_pattern = FALSE;		
2470		extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;		
2471		load_starting_address(instance);		
2472		#ifdef DBASE		
2473		if (lm_tmg_check_locked() != SUCCESS) {		
2474		start_time = lm_time();		
2475		while (lm_tmg_check_locked() != SUCCESS) {		
2476		if ((lm_time() - start_time) > TMC_LOCK_TIMEOUT) {		
2477		lm_queue_message(ERROR_MSG, "failed to lock Timing Generator");		
2478		instance->fatal_error = TRUE;		
2479		return(FAILURE);		
2480		}		
2481		}		
2482		#endif		
2483		/* Check if the DAB configuration has changed. */		
2484		if (modeler_error.error == TRUE) {		
2485		modeler_error.error = FALSE;		
2486		if ((modeler_error.pac_lane_errors != 0)		
2487		(modeler_error.tmg_error == TRUE)		
2488		(modeler_error.unknown_source_of_interrupt == TRUE)) {		
2489		fatal_hardware_error_encountered = TRUE;		



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
22/172

```

LINE #          SOURCE TEXT
2521      get_fatal_hardware_message();
2522      instance->fatal_error = TRUE;
2523      return(FAILURE);
2524  }
2525
2526      (modeler_error.dab_change) {
2527      reconfigure_dab();
2528  }
2529  else {
2530      /* Check PEL errors */
2531      if (modeler_error.pel_error_list != 0) {
2532          if (check_pel_errors(instance) == FAILURE) {
2533              instance->fatal_error = TRUE;
2534              return(FAILURE);
2535          }
2536      }
2537  }
2538
2539  if (extra_def_ptr->dab_ok == FALSE) {
2540      lm_queue_message(ERROR_MSG, "Device Adapter was removed");
2541      instance->fatal_error = TRUE;
2542      return(FAILURE);
2543  }
2544
2545  if ("changed_dac" == TRUE) {
2546      while (read_timer1() == FAILURE)
2547          "changed_dac" = FALSE;
2548  }
2549
2550  while (read_timer0(telapsed) == FAILURE)
2551      ;
2552
2553  #ifdef DRAKE
2554  /* Start pattern play */
2555  if (lm_tmng_initiate_play() == FAILURE) {
2556      lm_queue_message(ERROR_MSG, "failed to initiate pattern play");
2557      instance->fatal_error = TRUE;
2558      return(FAILURE);
2559  }
2560  #endif
2561
2562  #ifdef MODELER
2563  if (lm_tmng_complete_play(timeout) == FAILURE) {
2564      lm_queue_message(ERROR_MSG, "failed to complete pattern play");
2565      instance->fatal_error = TRUE;
2566      return(FAILURE);
2567  }
2568  #else
2569  #ifdef DIRECTOON
2570  /* Poll CPU status for TNG interrupt */
2571  start_time = lm_time();
2572  while (cpu_tmng_interrupt_status() == FALSE) {
2573      if ((lm_time() - start_time) > timeout) {
2574          DPRINTF(("play_ptrn_seq: TIMEOUT\n"));
2575          lm_tmng_abort_play();
2576          DPRINTF(("finished aborting play\n"));
2577          tmngptr->pattern_intr_enable = 0;
2578          lm_queue_message(ERROR_MSG, "timeout during pattern play");
2579          instance->fatal_error = TRUE;
2580          return(FAILURE);
2581      }
2582  }
2583  tmngptr->pattern_intr_enable = 0;
2584  #else
2585  /* Do nothing if it's data base access */
2586  #endif
2587  #endif
2588
2589  if (modeler_error.error == TRUE) {
2590      modeler_error.error = FALSE;
2591
2592      if ((modeler_error.pac_lase_errors != 0) ||
2593          (modeler_error.tmng_error == TRUE) ||
2594          (modeler_error.unknown_source_of_interrupt == TRUE)) {
2595          fatal_hardware_error_accounted = TRUE;
2596          get_fatal_hardware_message();
2597          instance->fatal_error = TRUE;
2598          return(FAILURE);
2599      }
2600  }
2601
2602  if (modeler_error.dab_change) {
2603      reconfigure_dab();
2604      replay_pattern = TRUE;
2605  }
2606  else {
2607      /* Check PEL errors */
2608      if (modeler_error.pel_error_list != 0) {
2609          if (check_pel_errors(instance) == FAILURE) {
2610              instance->fatal_error = TRUE;
2611              return(FAILURE);
2612          }
2613      }
2614  }
2615
2616  if (extra_def_ptr->dab_ok == FALSE) {
2617      lm_queue_message(ERROR_MSG, "Device Adapter was removed");
2618      instance->fatal_error = TRUE;
2619      return(FAILURE);
2620  }
2621
2622  if (replay_pattern == TRUE) {
2623      /* The DAB configuration was changed but, this device was not
2624      * invalidated by reconfigure_dab() (i.e. extra_def_ptr->dab_ok is still
2625      * TRUE). This could mean that this particular device is not touched
2626      * at all or that this device was removed but inserted back in.
2627      * Play the pattern again to take care of the second condition.
2628      */
2629      if (play_ptrn_seq(instance, timeout, changed_dac) == FAILURE) {
2630          instance->fatal_error = TRUE;
2631          return(FAILURE);
2632      }
2633  }
2634
2635  return(SUCCESS);
2636
2637  }
2638
2639  return_all_ptrn_block(instance);

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 23/173
LINE #		SOURCE TEXT		
2641		INSTANCE_INFO *instance;		
2642		{		
2643		/* Returns all of the pattern block used by this instance. This routine		
2644		/* looks at the seq_start_addr (for instance) or var_seq_addr (for fault)		
2645		/* to find the beginning of the pattern block chain.		
2646		*/		
2647		LANE_ADDR_INFO *lane_ptr;		
2648		EXTRA_DEVICE_SPEC *extra_def_ptr;		
2649		u_long block_addr;		
2650		u_short next_block_number;		
2651		u_char lane;		
2652		u_char fb_block_count;		
2653		char i;		
2654		{		
2655		DPRINTF(("Inside returns_all_ptr_block\n"));		
2656		extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;		
2657		for (lane = 0; lane < MAX_LANE_COUNT; ++lane) {		
2658		{		
2659		lane_ptr = (LANE_ADDR_INFO *)instance->lane_addr[lane];		
2660		if (extra_def_ptr->lane_used & (1 << lane)) {		
2661		{		
2662		if (instance->is_fault == TRUE) {		
2663		block_addr = instance->var_seq_addr[lane];		
2664		instance->var_seq_addr[lane] = 0;		
2665		}		
2666		else {		
2667		block_addr = instance->seq_start_addr[lane];		
2668		instance->seq_start_addr[lane] = 0;		
2669		}		
2670		/* This happens if we fail when we call new_block() */		
2671		if (block_addr == 0) {		
2672		printf("block_addr is 0\n");		
2673		continue;		
2674		}		
2675		next_block_number = read_branch_table_content(block_addr);		
2676		while (next_block_number != 0) {		
2677		{		
2678		DPRINTF(("next_block_number: %08x\n", next_block_number));		
2679		release_block(block_addr, lane);		
2680		block_addr = (block_addr & LANE_ADDR_MASK) +		
2681		(next_block_number << BLOCK_NUMBER_SHIFT);		
2682		}		
2683		next_block_number = read_branch_table_content(block_addr);		
2684		release_block(block_addr, lane);		
2685		/* If the pattern sequence has feedback, release the remaining		
2686		/* feedback blocks.		
2687		*/		
2688		if ((instance->fb_block_size[lane] != 0) &&		
2689		(instance->is_fault == FALSE)) {		
2690		{		
2691		DPRINTF(("releasing remaining feedback blocks\n"));		
2692		block_addr = instance->fb_block_addr[lane] + BLOCK_ADDR_INC;		
2693		fb_block_count = instance->fb_block_size[lane] - 1;		
2694		instance->fb_block_addr[lane] = 0;		
2695		instance->fb_block_size[lane] = 0;		
2696		if (block_addr != BLOCK_ADDR_INC) {		
2697		for (i = 0; i < fb_block_count; ++i) {		
2698		{		
2699		release_block(block_addr, lane);		
2700		block_addr += BLOCK_ADDR_INC;		
2701		}		
2702		}		
2703		}		
2704		}		
2705		}		
2706		}		
2707		}		
2708		}		
2709		}		
2710		}		
2711		}		
2712		}		
2713		}		
2714		}		
2715		}		
2716		}		
2717		}		
2718		}		
2719		}		
2720		}		
2721		}		
2722		#ifdef DEBUG		
2723		{		
2724		print_pin_changes(instance);		
2725		INSTANCE_INFO *instance;		
2726		{		
2727		PIN_INFO *pin_info;		
2728		short pin_number;		
2729		{		
2730		DPRINTF(("DATA pin changes: \n"));		
2731		pin_number = instance->first_data_pin_index;		
2732		while (pin_number != -1) {		
2733		{		
2734		pin_info = instance->pin_info_table[pin_number];		
2735		DPRINTF((" pin: %d time: %d", pin_number, pin_info->sim_time));		
2736		switch (pin_info->old_filtered) {		
2737		{		
2738		case LOGIC_0:		
2739		DPRINTF((" logval: 0\n"));		
2740		break;		
2741		case LOGIC_1:		
2742		DPRINTF((" logval: 1\n"));		
2743		break;		
2744		case LOGIC_20:		
2745		DPRINTF((" logval: 20\n"));		
2746		break;		
2747		case LOGIC_21:		
2748		DPRINTF((" logval: 21\n"));		
2749		break;		
2750		case LOGIC_U:		
2751		DPRINTF((" logval: U\n"));		
2752		break;		
2753		case LOGIC_S0:		
2754		DPRINTF((" logval: S0\n"));		
2755		break;		
2756		case LOGIC_S1:		
2757		DPRINTF((" logval: S1\n"));		
2758		break;		
2759		default:		
2760		{		
		break;		
		}		
		pin_number = pin_info->next_input_pin_index;		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
24/174

```

LINE #          SOURCE TEXT
2761      }
2762
2763      DPRINTF(("EVAL pin changes: \n"));
2764      pin_number = instance->first_eval_pin_index;
2765      while (pin_number != -1) {
2766          pin_info = instance->pin_info_table[pin_number];
2767          DPRINTF((" SW: ed stime: ed", pin_number, pin_info->sin_time));
2768          switch (pin_info->old_filtered) {
2769              case LOGIC_0:
2770                  DPRINTF((" lagval: 0\n"));
2771                  break;
2772              case LOGIC_1:
2773                  DPRINTF((" lagval: 1\n"));
2774                  break;
2775              case LOGIC_20:
2776                  DPRINTF((" lagval: 20\n"));
2777                  break;
2778              case LOGIC_21:
2779                  DPRINTF((" lagval: 21\n"));
2780                  break;
2781              case LOGIC_U:
2782                  DPRINTF((" lagval: U\n"));
2783                  break;
2784              case LOGIC_S0:
2785                  DPRINTF((" lagval: S0\n"));
2786                  break;
2787              case LOGIC_S1:
2788                  DPRINTF((" lagval: S1\n"));
2789                  break;
2790              default:
2791                  break;
2792          }
2793          pin_number = pin_info->next_input_pin_index;
2794      }
2795
2796      DPRINTF(("STORE pin changes: \n"));
2797      pin_number = instance->first_store_pin_index;
2798      while (pin_number != -1) {
2799          pin_info = instance->pin_info_table[pin_number];
2800          DPRINTF((" SW: ed stime: ed", pin_number, pin_info->sin_time));
2801          switch (pin_info->old_filtered) {
2802              case LOGIC_0:
2803                  DPRINTF((" lagval: 0\n"));
2804                  break;
2805              case LOGIC_1:
2806                  DPRINTF((" lagval: 1\n"));
2807                  break;
2808              case LOGIC_20:
2809                  DPRINTF((" lagval: 20\n"));
2810                  break;
2811              case LOGIC_21:
2812                  DPRINTF((" lagval: 21\n"));
2813                  break;
2814              case LOGIC_U:
2815                  DPRINTF((" lagval: U\n"));
2816                  break;
2817              case LOGIC_S0:
2818                  DPRINTF((" lagval: S0\n"));
2819                  break;
2820              case LOGIC_S1:
2821                  DPRINTF((" lagval: S1\n"));
2822                  break;
2823              default:
2824                  break;
2825          }
2826          pin_number = pin_info->next_input_pin_index;
2827      }
2828  }
2829
2830  #endif
2831
2832  fill_in_extra_data(def_ptr, dab_info_index);
2833  DEVICE_SPEC *def_ptr;
2834  char
2835      {
2836      PIN_SPEC
2837      EXTRA_DEVICE_SPEC
2838      DAB_INFO
2839      UWB_OFFSET
2840      u_short
2841      u_short
2842      u_char
2843      u_char
2844      u_char
2845      u_char
2846
2847      extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
2848      dab_ptr
2849          = dab_list[dab_info_index];
2850
2851      for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
2852          if (dab_ptr->lane_used[lane0]) {
2853              extra_def_ptr->lane_used[lane0] = 1 << lane0;
2854          }
2855      }
2856
2857      extra_def_ptr->has_to_store = FALSE;
2858      extra_def_ptr->ident_inputs = (PTRN_BITS_LONGWORD *)
2859          DCALLOC((unsigned)dab_ptr->unit_count,
2860              (unsigned)sizeof(PTRN_BITS_LONGWORD));
2861      if (extra_def_ptr->ident_inputs == NULL)
2862          return(FAILURE);
2863
2864      extra_def_ptr->ident_outputs = (PTRN_BITS_LONGWORD *)
2865          DCALLOC((unsigned)dab_ptr->unit_count,
2866              (unsigned)sizeof(PTRN_BITS_LONGWORD));
2867      if (extra_def_ptr->ident_outputs == NULL)
2868          return(FAILURE);
2869
2870      extra_def_ptr->ident_ios = (PTRN_BITS_LONGWORD *)
2871          DCALLOC((unsigned)dab_ptr->unit_count,
2872              (unsigned)sizeof(PTRN_BITS_LONGWORD));
2873      if (extra_def_ptr->ident_ios == NULL)
2874          return(FAILURE);
2875
2876      extra_def_ptr->ident_R1 = (PTRN_BITS_LONGWORD *)
2877          DCALLOC((unsigned)dab_ptr->unit_count,
2878              (unsigned)sizeof(PTRN_BITS_LONGWORD));
2879      if (extra_def_ptr->ident_R1 == NULL)
2880          return(FAILURE);

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 25/175
LINE #	SOURCE TEXT			
2881	extra_def_ptr->ident_R1 = (PTRN_BITS_LONGWORD *)			
2882	DCALLOC((unsigned)dab_ptr->unit_count,			
2883	(unsigned)sizeof(PTRN_BITS_LONGWORD));			
2884	if (extra_def_ptr->ident_R1 == NULL)			
2885	return(FAILURE);			
2886				
2887	extra_def_ptr->ident_store = (PTRN_BITS_LONGWORD *)			
2888	DCALLOC((unsigned)dab_ptr->unit_count,			
2889	(unsigned)sizeof(PTRN_BITS_LONGWORD));			
2890	if (extra_def_ptr->ident_store == NULL)			
2891	return(FAILURE);			
2892				
2893	pin_count = dab_ptr->pin_cnt;			
2894	pin_spec_ptr = &def_ptr->pin_table[0];			
2895	for (pin_number = 0; pin_number < pin_count; ++pin_number) {			
2896	if ((pin_spec_ptr->direction == NONE)			
2897	(pin_spec_ptr->direction == POWER)			
2898	(pin_spec_ptr->direction == GROUND)			
2899	(pin_spec_ptr->direction == NC)) {			
2900	++pin_spec_ptr;			
2901	continue;			
2902				
2903				
2904				
2905	word_ptr = &pin_spec_ptr->word;			
2906	unitno = word_ptr->unitno;			
2907	wordno = word_ptr->wordno;			
2908	bitno = word_ptr->bitno;			
2909				
2910	if (pin_spec_ptr->pin_class == STORE) {			
2911	set_ptrn_bit(&extra_def_ptr->ident_store [unitno], wordno, bitno);			
2912				
2913	if (pin_spec_ptr->direction == IO)			
2914	extra_def_ptr->has_io_store = TRUE;			
2915				
2916				
2917	switch (pin_spec_ptr->direction) {			
2918	case IN:			
2919	set_ptrn_bit(&extra_def_ptr->ident_inputs [unitno], wordno, bitno);			
2920	break;			
2921	case OUT:			
2922	set_ptrn_bit(&extra_def_ptr->ident_outputs [unitno], wordno, bitno);			
2923	break;			
2924	case IO:			
2925	set_ptrn_bit(&extra_def_ptr->ident_ioa [unitno], wordno, bitno);			
2926	break;			
2927	default:			
2928	break;			
2929				
2930				
2931	switch (pin_spec_ptr->clk_format) {			
2932	case R1:			
2933	set_ptrn_bit(&extra_def_ptr->ident_R1 [unitno], wordno, bitno);			
2934	break;			
2935	case R0:			
2936	set_ptrn_bit(&extra_def_ptr->ident_R2 [unitno], wordno, bitno);			
2937	break;			
2938	default:			
2939	break;			
2940				
2941				
2942	++pin_spec_ptr;			
2943				
2944				
2945	return(SUCCESS);			
2946				
2947				
2948	increment_unit_addr(unit_addr, unit_count, unit_count_per_lane)			
2949	u_long unit_addr[];			
2950	u_short unit_count;			
2951	u_short unit_count_per_lane;			
2952	{			
2953	u_long next_block_addr;			
2954	u_long max_block_addr;			
2955	u_short next_block_number;			
2956	u_short addr_inc_per_lane;			
2957	u_char unitno;			
2958				
2959	addr_inc_per_lane = unit_count_per_lane * PTRN_ADDR_INC;			
2960				
2961	for (unitno = 0; unitno < unit_count; ++unitno) {			
2962	max_block_addr = (unit_addr[unitno] & BLOCK_START_MASK) + BLOCK_ADDR_INC;			
2963				
2964	if ((unit_addr[unitno] + addr_inc_per_lane) >= max_block_addr) {			
2965	next_block_number = read_branch_table_content(unit_addr[unitno]);			
2966				
2967	next_block_addr = (unit_addr[unitno] & LANE_ADDR_MASK) +			
2968	(next_block_number << BLOCK_NUMBER_SHIFT);			
2969				
2970	unit_addr[unitno] = next_block_addr +			
2971	unit_addr[unitno] + addr_inc_per_lane - max_block_addr;			
2972				
2973	else {			
2974	unit_addr[unitno] += addr_inc_per_lane;			
2975				
2976				
2977				
2978				
2979				
2980				
2981	set_feedback_branch(instance, branch_offset)			
2982	INSTANCE_INFO *instance;			
2983	short branch_offset;			
2984	{			
2985				
2986	DEVICE_SPEC *def;			
2987	DAB_INFO *dab_ptr;			
2988	u_long addr;			
2989	u_long temp;			
2990	u_char lanesno;			
2991				
2992	def = instance->definition;			
2993	dab_ptr = dab_list(instance->dab_info_index);			
2994				
2995				
2996	for (lanesno = 0; lanesno < MAX_LANE_COUNT; ++lanesno) {			
2997	if (dab_ptr->lane_used[lanesno]) {			
2998	addr = instance->fb_block_addr[lanesno] +			
2999	branch_offset + PTRN_ADDR_INC + LANE_SEGMENT_C_OFFSET;			
3000				

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	26/176

```

LINE # SOURCE TEXT
3001 DPRINTF(("set feedback branch at addr: %08X\n", addr));
3002
3003 temp = read_loc_long((u_long *)addr);
3004
3005 if (def->fb_type == FEEDBACK_RISE)
3006     temp = temp & PEL_BRANCH_MASK | BRANCH_RISE << PEL_BRANCH_SHIFT;
3007 else
3008     temp = temp & PEL_BRANCH_MASK | BRANCH_FALL << PEL_BRANCH_SHIFT;
3009
3010 write_loc_long((u_long *)addr, temp);
3011
3012 }
3013
3014 }
3015
3016 /-----/
3017
3018 read_magic_full_sample_reg(instance, full_value_ptr)
3019 INSTANCE_INFO *instance,
3020 FULL_VALUE *full_value_ptr,
3021 {
3022
3023 #ifdef DBASE
3024     DAB_INFO *dab_ptr;
3025     u_long addr;
3026     register u_long magic_addr;
3027     u_char total_unit;
3028     u_char unitno;
3029     u_char lane_no;
3030     u_char slotno;
3031     register u_char wordno;
3032     register PTN_BITS *data_ptr;
3033     register PTN_BITS *hiz_ptr;
3034     register PTN_BITS *unk_ptr;
3035
3036     DPRINTF(("inside read_magic_full_sample_reg\n"));
3037
3038     dab_ptr = dab_list(instance->dab_info_index);
3039
3040     data_ptr = full_value_ptr->data;
3041     hiz_ptr = full_value_ptr->hiz;
3042     unk_ptr = full_value_ptr->unknown;
3043
3044     total_unit = dab_ptr->unit_count;
3045     for (unitno = 0; unitno < total_unit; ++unitno) {
3046         lane_no = dab_ptr->unit_location[unitno].lane_no;
3047         slotno = dab_ptr->unit_location[unitno].slot_no;
3048
3049         addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +
3050             LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC;
3051
3052         magic_addr = addr + PEL_MC_0_OFFSET +
3053             (MAX_SHORT_WORD_COUNT - 1) * PEL_MC_ADDR_INC;
3054         for (wordno = 0; wordno < MAX_SHORT_WORD_COUNT; ++wordno) {
3055             /* MAGIC: 0 -> pin 15...0
3056              * PTN_PTR->word[0] -> pin 77...64
3057              */
3058             data_ptr->word[wordno] = read_loc_long((u_long *)
3059                 (magic_addr + MC_VALUE_SAMPLE_REG_OFFSET));
3060             hiz_ptr->word[wordno] = read_loc_long((u_long *)
3061                 (magic_addr + MC_HIZ_SAMPLE_REG_OFFSET));
3062             unk_ptr->word[wordno] = read_loc_long((u_long *)
3063                 (magic_addr + MC_U_SAMPLE_REG_OFFSET));
3064             magic_addr += PEL_MC_ADDR_INC;
3065         }
3066
3067         DPRINTF((" DATA: %08X %08X %08X\n",
3068             ((PTN_BITS_LONGWORD *)data_ptr->word[0]),
3069             ((PTN_BITS_LONGWORD *)data_ptr->word[1]),
3070             ((PTN_BITS_LONGWORD *)data_ptr->word[2]));
3071
3072         DPRINTF((" HIZ: %08X %08X %08X\n",
3073             ((PTN_BITS_LONGWORD *)hiz_ptr->word[0]),
3074             ((PTN_BITS_LONGWORD *)hiz_ptr->word[1]),
3075             ((PTN_BITS_LONGWORD *)hiz_ptr->word[2]));
3076
3077         DPRINTF((" UNK: %08X %08X %08X\n",
3078             ((PTN_BITS_LONGWORD *)unk_ptr->word[0]),
3079             ((PTN_BITS_LONGWORD *)unk_ptr->word[1]),
3080             ((PTN_BITS_LONGWORD *)unk_ptr->word[2]));
3081
3082         ++data_ptr;
3083         ++hiz_ptr;
3084         ++unk_ptr;
3085     }
3086 #else
3087     DAB_INFO *dab_ptr;
3088     DAB_OFFSET *dab_ptr;
3089     char line[80];
3090     u_long pin_number;
3091     u_short total_pin;
3092     u_char pin_value;
3093     u_char unitno;
3094     u_char wordno;
3095     u_char bitno;
3096
3097     DPRINTF(("inside read_magic_full_sample_reg()\n"));
3098
3099     if (is_measure_delay == TRUE) {
3100         dab_ptr = dab_list(instance->dab_info_index);
3101         total_pin = dab_ptr->unit_count * 80;
3102
3103         for (pin_number = 0; pin_number < total_pin; ++pin_number) {
3104             if (tm_result_array(pin_number).set_by_user == TRUE) {
3105
3106                 unitno = pin_to_short_offset(pin_number);
3107                 wordno = unitno->wordno;
3108                 bitno = unitno->bitno;
3109
3110                 set_pin_value(full_value_ptr, unitno, wordno, bitno,
3111                     tm_result_array(pin_number).steady_state_full_value);
3112             }
3113         }
3114     }
3115 }
3116
3117 }
3118
3119 }
3120

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89 PAGE #  
TIME 6:14:53 pm 27/177

```

LINE # SOURCE TEXT
3121     return;
3122 }
3123
3124 DPRINTF(("enter starting pin_number: %d"),
3125         gets(line),
3126         sscanf(line, "%d", &pin_number));
3127
3128 /* assume that the pin_number is sequential */
3129 DPRINTF(("enter pin value on at a time, and with 'X' (legal val: 0, 1, 20, 21, U)\n"));
3130
3131 while (1) {
3132     DPRINTF(("pin value: "));
3133     gets(line);
3134     if (strcmp(line, "0") == 0)
3135         pin_value = LOGIC_0;
3136     else if (strcmp(line, "1") == 0)
3137         pin_value = LOGIC_1;
3138     else if (strcmp(line, "20") == 0)
3139         pin_value = LOGIC_20;
3140     else if (strcmp(line, "21") == 0)
3141         pin_value = LOGIC_21;
3142     else if (strcmp(line, "U") == 0)
3143         pin_value = LOGIC_U;
3144     else if (strcmp(line, "X") == 0)
3145         break;
3146     else {
3147         DPRINTF(("illegal value. ignored\n"));
3148         continue;
3149     }
3150 }
3151
3152 word_ptr = &pa_to_short_offset[pin_number];
3153 unitno = word_ptr->unitno;
3154 wordno = word_ptr->wordno;
3155 bitno = word_ptr->bitno;
3156
3157 set_pin_value(full_value_ptr, unitno, wordno, bitno, pin_value);
3158
3159 ++pin_number;
3160 }
3161 }
3162 }
3163 }
3164 }
3165 }
3166 }
3167
3168 /* ARGUSED */
3169 read_magic_timing_sample_reg(instance, ptrs_long_ptr, current_time)
3170 INSTANCE_INFO *instance,
3171 PTRN_BITS_LONGWORD *ptrs_long_ptr,
3172 u_short *current_time;
3173 {
3174     #ifndef DBASE
3175     DAB_INFO *dab_ptr;
3176     u_long *u_long;
3177     u_long magic_addr;
3178     u_char total_unit;
3179     u_char unitno;
3180     u_char lane0;
3181     u_char slotno;
3182     u_char wordno;
3183     PTRN_BITS *ptrs_ptr;
3184
3185     DPRINTF(("inside read_magic_timing_sample_reg\n"));
3186
3187     dab_ptr = dab_list(instance->dab_info_index);
3188
3189     ptrs_ptr = (PTRN_BITS *)ptrs_long_ptr;
3190     total_unit = dab_ptr->unit_count;
3191     for (unitno = 0; unitno < total_unit; ++unitno) {
3192         lane0 = dab_ptr->unit_location(unitno).lane_no;
3193         slotno = dab_ptr->unit_location(unitno).slot_no;
3194
3195         addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC +
3196               LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC;
3197
3198         magic_addr = addr + PEL_MC_0_OFFSET + MC_TIMING_SAMPLE_REC_OFFSET +
3199                     (MAX_SHORT_WORD_COUNT - 1) * PEL_MC_ADDR_INC;
3200         for (wordno = 0; wordno < MAX_SHORT_WORD_COUNT; ++wordno) {
3201             /* MAGIC 0 -----> pin 15..0
3202              * ptrs_ptr->word[0] -----> pin 71..64
3203              */
3204             ptrs_ptr->word[wordno] = read_loc_long((u_long *)magic_addr);
3205             magic_addr += PEL_MC_ADDR_INC;
3206         }
3207
3208         DPRINTF(("DATA: 0001 0002 0003\n",
3209                 ((PTRN_BITS_LONGWORD *)ptrs_ptr)->word[0],
3210                 ((PTRN_BITS_LONGWORD *)ptrs_ptr)->word[1],
3211                 ((PTRN_BITS_LONGWORD *)ptrs_ptr)->word[2]));
3212
3213         ++ptrs_ptr;
3214     }
3215
3216     #else
3217     DAB_INFO *dab_ptr;
3218     DAB_O *dab_o;
3219     char line[80];
3220     u_long pin_number;
3221     u_short total_pin;
3222     u_char unitno;
3223     u_char wordno;
3224     u_char bitno;
3225
3226     DPRINTF(("inside read_magic_timing_sample_reg()\n"));
3227
3228     if (in_measure_delay == TRUE) {
3229         dab_ptr = dab_list(instance->dab_info_index);
3230         total_pin = dab_ptr->unit_count * 80;
3231         for (pin_number = 0; pin_number < total_pin; ++pin_number) {
3232             if (tm_result_array[pin_number].set_by_user == TRUE) {
3233                 word_ptr = &pa_to_short_offset[pin_number];
3234                 unitno = word_ptr->unitno;
3235                 wordno = word_ptr->wordno;
3236                 bitno = word_ptr->bitno;
3237             }
3238         }
3239     }
3240

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	28/178

```

LINE # SOURCE TEXT
3241 if (tm_result_array[pin_number].expected_time <= current_time) {
3242 if (tm_result_array[pin_number].two_state_value == LOGIC_0) {
3243 reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3244 wordno, bitno);
3245 }
3246 else {
3247 set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3248 wordno, bitno);
3249 }
3250 }
3251 else {
3252 if (tm_result_array[pin_number].two_state_value == LOGIC_0) {
3253 set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3254 wordno, bitno);
3255 }
3256 else {
3257 reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno),
3258 wordno, bitno);
3259 }
3260 }
3261 }
3262 }
3263 return;
3264 }
3265
3266 DPRINTF(("enter starting pin number: \n"));
3267 gets(line);
3268 sscanf(line, "%d", &pin_number);
3269
3270 /* assume that the BMS pin number is sequential */
3271 DPRINTF(("enter pin value on at a time, and with 'X' (legal val: 0, 1)\n"));
3272
3273 while (1) {
3274
3275     uwb_ptr = &tm_ptrn_offset[pin_number];
3276     uwb_ptr = uwb_ptr->unitno;
3277     wordno = uwb_ptr->wordno;
3278     bitno = uwb_ptr->bitno;
3279
3280     DPRINTF(("pin value: "));
3281     gets(line);
3282
3283     if (strcmp(line, "0") == 0)
3284         reset_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno), wordno, bitno);
3285     else if (strcmp(line, "1") == 0)
3286         set_ptrn_bit(&((PTRN_BITS *)ptrn_long_ptr)(unitno), wordno, bitno);
3287     else if (strcmp(line, "X") == 0)
3288         break;
3289     else {
3290         DPRINTF(("illegal value. ignored\n"));
3291         continue;
3292     }
3293     ++pin_number;
3294 }
3295
3296 }
3297 #endif
3298
3299
3300
3301 clear_ptrn_bits(ptrn_ptr, total_unit)
3302 char *ptrn_ptr
3303 char *total_unit
3304 {
3305     char *memset();
3306     (void)memset(ptrn_ptr, 0, (int)(total_unit * sizeof(PTRN_BITS)));
3307 }
3308
3309 copy_ptrn_bits(source, dest, total_unit)
3310 char *source;
3311 char *dest;
3312 u_char total_unit;
3313 {
3314     char *memcpy();
3315     (void)memcpy(dest, source, (int)(total_unit * sizeof(PTRN_BITS)));
3316 }
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
29/179

```

LINE #          SOURCE TEXT
3361          extra_def_ptr->source_reg) != SUCCESS) {
3362      lm_queue_message(ERROR_MSG, "Timing Generator error: set frequency");
3363      return(FAILURE);
3364  }
3365
3366  /* Set the mode to EARLY SAMPLE and the EDGE 7 threshold to 255,
3367   * in order to get a sample pulse.
3368   * These values will be overriden if we are doing timing measurement.
3369  */
3370  extra_def_ptr->edge_setting[6] = 255;
3371
3372  if (lm_tmg_set_sample_trigger_mode(
3373      (u_long)EARLYSAMPLETRIGGERMODE) == FAILURE) {
3374      lm_queue_message(ERROR_MSG, "Timing Generator error: set sample trigger mode");
3375      return(FAILURE);
3376  }
3377
3378  return(SUCCESS);
3379  }
3380
3381  load_starting_address(instance)
3382  INSTANCE_INFO *instance;
3383  {
3384      /* Load the starting pattern address on each lane and
3385       * also load the clock frequency register.
3386       */
3387      EXTRA_DEVICE_SPEC *extra_def_ptr;
3388      DAB_INFO *dab_ptr;
3389      u_long addr;
3390      u_char lane0;
3391      u_short blocknum;
3392      u_char clock_speed_reg;
3393
3394      extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;
3395
3396      if (extra_def_ptr->actual_phy_clock_period < PAC_FREQUENCY_THRESHOLD)
3397          clock_speed_reg = 1;
3398      else
3399          clock_speed_reg = 0;
3400
3401      dab_ptr = dab_list(instance->dab_info_index);
3402
3403      /* Tell the TME which lane we are playing to */
3404      lm_tmg_lane_select(dab_ptr->ident_lane);
3405
3406      for (lane0 = 0; lane0 < MAX_LANE_COUNT; ++lane0) {
3407          if (dab_ptr->lane_used[lane0]) {
3408              blocknum = ptob(instance->seq_start_addr[lane0]);
3409              addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC +
3410                  LANE_PAC_START_OFFSET;
3411
3412              /* set the starting address */
3413              write_loc_long((u_long *))(addr + PAC_BRANCH_ADDR_REG_OFFSET),
3414                  (u_long)blocknum);
3415
3416              /* set the clock speed register */
3417              write_loc_long((u_long *))(addr + PAC_CLOCK_SPEED_REG_OFFSET),
3418                  (u_long)clock_speed_reg);
3419          }
3420      }
3421  }
3422
3423  #ifdef MODELER
3424  ac_delay()
3425  {
3426  }
3427  #endif
3428
3429  set_edge_and_sample_setting(extra_def_ptr, resolution, sample_threshold)
3430  EXTRA_DEVICE_SPEC *extra_def_ptr;
3431  u_long resolution;
3432  u_long sample_threshold;
3433  {
3434      #ifdef BRASS
3435          if (lm_tmg_set_edge_settings(extra_def_ptr->logical_clock_period,
3436              extra_def_ptr->edge_setting) != SUCCESS) {
3437              lm_queue_message(ERROR_MSG, "Timing Generator error: set edge settings");
3438              return(FAILURE);
3439          }
3440
3441          if (lm_tmg_set_rising_sample(resolution, sample_threshold) != SUCCESS) {
3442              lm_queue_message(ERROR_MSG, "Timing Generator error: set rising sample");
3443              return(FAILURE);
3444          }
3445
3446          if (lm_tmg_set_falling_sample(extra_def_ptr->falling_sample_reg) !=
3447              SUCCESS) {
3448              lm_queue_message(ERROR_MSG, "Timing Generator error: set falling sample");
3449              return(FAILURE);
3450          }
3451      }
3452  #endif
3453
3454  #ifdef TMR
3455      setup_timer(TMR_TIMER_VALUE);
3456      return(SUCCESS);
3457  }
3458
3459  calc_dab_voltage(def_ptr, dab_ptr)
3460  DEVICE_SPEC *def_ptr;
3461  DAB_INFO *dab_ptr;
3462  {
3463      EXTRA_DEVICE_SPEC *extra_def_ptr;
3464      u_short temp;
3465      u_short adjusted_val;
3466      u_short adjusted_vsh;
3467      u_short adjusted_vil;
3468      u_short adjusted_vih;
3469      u_short adjusted_vlth;
3470      u_short adjusted_vsth;
3471
3472      extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
3473
3474      if (dab_ptr->dut_vcc_measured < MIN_DUTVCC) {
3475          lm_queue_message(ERROR_MSG, "device VCC measured (%d mV) is too low, min allowed: %d mV",
3476              dab_ptr->dut_vcc_measured,
3477              MIN_DUTVCC);
3478          return(FAILURE);
3479      }
3480      if (dab_ptr->dut_vcc_measured > MAX_DUTVCC) {

```



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	30/180

```

LINE #          SOURCE TEXT
3481      lm_queue_message(ERROR_MSG, "device VCC measured (%d mV) is too high, max allowed: %d mV",
3482      dab_ptr->dut_vcc_measured,
3483      MAX_DUTVCC);
3484      return(FAILURE);
3485  }
3486
3487  if (within_tolerance(dab_ptr->dut_vcc_measured, def_ptr->vcc) == FALSE) {
3488      lm_queue_message(ERROR_MSG, "device VCC does not match, measured: %d mV specified: %d mV",
3489      dab_ptr->dut_vcc_measured, def_ptr->vcc);
3490      return(FAILURE);
3491  }
3492
3493  if (def_ptr->val > def_ptr->vcc / 5) {
3494      lm_queue_message(ERROR_MSG, "Val out of range. Specified: %d mV range: (%d mV - %d mV)",
3495      def_ptr->val, 0, def_ptr->vcc / 5);
3496      return(FAILURE);
3497  }
3498
3499  if ((def_ptr->vsh < def_ptr->vcc / 2) ||
3500      (def_ptr->vsh > def_ptr->vcc)) {
3501      lm_queue_message(ERROR_MSG, "Vsh out of range. Specified: %d mV range: (%d mV - %d mV)",
3502      def_ptr->vsh,
3503      def_ptr->vcc / 2,
3504      def_ptr->vcc);
3505      return(FAILURE);
3506  }
3507
3508  if (def_ptr->vth > def_ptr->vcc / 5) {
3509      lm_queue_message(ERROR_MSG, "Vth out of range. Specified: %d mV range: (%d mV - %d mV)",
3510      def_ptr->vth, 0, def_ptr->vcc / 5);
3511      return(FAILURE);
3512  }
3513
3514  if ((def_ptr->vath < def_ptr->vcc / 2) ||
3515      (def_ptr->vath > def_ptr->vcc)) {
3516      lm_queue_message(ERROR_MSG, "Vath out of range. Specified: %d mV range: (%d mV - %d mV)",
3517      def_ptr->vath,
3518      def_ptr->vcc / 2,
3519      def_ptr->vcc);
3520      return(FAILURE);
3521  }
3522
3523  if (def_ptr->vil > def_ptr->vcc) {
3524      lm_queue_message(ERROR_MSG, "vil out of range. Specified: %d mV range: (%d mV - %d mV)",
3525      def_ptr->vil,
3526      0,
3527      def_ptr->vcc);
3528      return(FAILURE);
3529  }
3530
3531  if (def_ptr->vih > def_ptr->vcc) {
3532      lm_queue_message(ERROR_MSG, "vih out of range. Specified: %d mV range: (%d mV - %d mV)",
3533      def_ptr->vih,
3534      0,
3535      def_ptr->vcc);
3536      return(FAILURE);
3537  }
3538
3539  adjusted_val = def_ptr->val * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3540  extra_def_ptr->val_dec_value =
3541      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, adjusted_val);
3542
3543  extra_def_ptr->private_val_dec_value =
3544      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, 0);
3545
3546  adjusted_vsh = def_ptr->vsh * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3547  extra_def_ptr->vsh_dec_value =
3548      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3549      dab_ptr->dut_vcc_measured,
3550      adjusted_vsh);
3551
3552  if (def_ptr->vsh + 400 > dab_ptr->dut_vcc_measured)
3553      temp = dab_ptr->dut_vcc_measured;
3554  else
3555      temp = def_ptr->vsh + 400;
3556
3557  extra_def_ptr->private_vsh_dec_value =
3558      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3559      dab_ptr->dut_vcc_measured,
3560      temp);
3561
3562  adjusted_vth = def_ptr->vth * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3563  extra_def_ptr->vth_dec_value =
3564      REG_VALUE(0, dab_ptr->dut_vcc_measured / 5, adjusted_vth);
3565
3566  adjusted_vath = def_ptr->vath * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3567  extra_def_ptr->vath_dec_value =
3568      REG_VALUE(dab_ptr->dut_vcc_measured / 2,
3569      dab_ptr->dut_vcc_measured,
3570      adjusted_vath);
3571
3572  adjusted_vil = def_ptr->vil * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3573  extra_def_ptr->vil_dec_value =
3574      REG_VALUE(0, dab_ptr->dut_vcc_measured, adjusted_vil);
3575
3576  adjusted_vih = def_ptr->vih * dab_ptr->dut_vcc_measured / def_ptr->vcc;
3577  extra_def_ptr->vih_dec_value =
3578      REG_VALUE(0, dab_ptr->dut_vcc_measured, adjusted_vih);
3579
3580  DPRINTF(("vih : %d mV -> %s", adjusted_vih,
3581      extra_def_ptr->vih_dec_value));
3582  DPRINTF(("vil : %d mV -> %s", adjusted_vil,
3583      extra_def_ptr->vil_dec_value));
3584  DPRINTF(("vath : %d mV -> %s", adjusted_vath,
3585      extra_def_ptr->vath_dec_value));
3586  DPRINTF(("vth : %d mV -> %s", adjusted_vth,
3587      extra_def_ptr->vth_dec_value));
3588  DPRINTF(("vsh : %d mV -> %s", adjusted_vsh,
3589      extra_def_ptr->vsh_dec_value));
3590  DPRINTF(("val : %d mV -> %s", adjusted_val,
3591      extra_def_ptr->val_dec_value));
3592  return(SUCCESS);
3593  }
3594
3595  within_tolerance(ref_value, value)
3596  u_short ref_value;
3597  u_short value;
3598  {
3599      /* Return TRUE if "value" is <= MAX_PERCENT_TOLERANCE percent from
3600      * "ref_value", else return FALSE.

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	31/181

```

LINE #          SOURCE TEXT
3601 //
3602
3603 if ((value < ref_value - MAX_PERCENT_TOLERANCE * ref_value / 100) ||
3604     (value > ref_value + MAX_PERCENT_TOLERANCE * ref_value / 100))
3605     return SK;
3606
3607 return(TRUE);
3608 }
3609
3610 verify_soft_drive_current(def_ptr, dab_ptr)
3611 DEVICE_SPEC *def_ptr;
3612 DAB_INFO *dab_ptr;
3613 {
3614     PIN_SPEC *pin_spec_ptr;
3615     total_current;
3616     pin_count;
3617     u_short max_pin_count;
3618     u_short soft_drive_low_count[MAX_LANE_COUNT * MAX_SLOT_COUNT][4];
3619     u_short soft_drive_high_count[MAX_LANE_COUNT * MAX_SLOT_COUNT][4];
3620     pinno;
3621     unitno;
3622     u_char palno;
3623     u_char temp;
3624     u_char current_limit_exceeded = FALSE;
3625
3626     for (palno = 0; palno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++palno) {
3627         for (temp = 0; temp < 4; ++temp) {
3628             soft_drive_low_count[palno][temp] = 0;
3629             soft_drive_high_count[palno][temp] = 0;
3630         }
3631     }
3632
3633     pin_count = def_ptr->pin_cnt;
3634     max_pin_count = dab_ptr->unit_count * MAX_PIN_PER_UNIT;
3635
3636     unitno = ps_to_ahort_offset(pin_count - 1).unitno;
3637     palno = dab_ptr->unit_location(unitno).lane_no * MAX_SLOT_COUNT +
3638             dab_ptr->unit_location(unitno).slot_no;
3639
3640     for (pinno = pin_count; pinno < max_pin_count; ++pinno) {
3641         /* These pins are not specified but are not in the device.h
3642            * because the pin_table in the device.h only goes up to
3643            * the largest pin the user specifies.
3644            */
3645         soft_drive_low_count[palno][0] += 1;
3646     }
3647
3648     pin_spec_ptr = def_ptr->pin_table[0];
3649     for (pinno = 0; pinno < pin_count; ++pinno) {
3650         unitno = ps_to_ahort_offset(pinno).unitno;
3651         palno = dab_ptr->unit_location(unitno).lane_no * MAX_SLOT_COUNT +
3652                 dab_ptr->unit_location(unitno).slot_no;
3653
3654         if (pin_spec_ptr->direction == NONE) {
3655             /* These pins will be soft-driven LOW with minimum current */
3656             soft_drive_low_count[palno][0] += 1;
3657             ++pin_spec_ptr;
3658             continue;
3659         }
3660
3661         if ((pin_spec_ptr->direction == POWER) ||
3662             (pin_spec_ptr->direction == GROUND) ||
3663             (pin_spec_ptr->direction == MC)) {
3664             ++pin_spec_ptr;
3665             continue;
3666         }
3667
3668         /* SDLEN[3-0] */
3669         temp = pin_spec_ptr->s_drive_hi;
3670         if (temp & 0x8)
3671             soft_drive_high_count[palno][3] += 1;
3672         if (temp & 0x4)
3673             soft_drive_high_count[palno][2] += 1;
3674         if (temp & 0x2)
3675             soft_drive_high_count[palno][1] += 1;
3676         if (temp & 0x1)
3677             soft_drive_high_count[palno][0] += 1;
3678
3679         /* SDLEN[3-0]S */
3680         temp = pin_spec_ptr->s_drive_low;
3681         if (temp & 0x8)
3682             soft_drive_low_count[palno][3] += 1;
3683         if (temp & 0x4)
3684             soft_drive_low_count[palno][2] += 1;
3685         if (temp & 0x2)
3686             soft_drive_low_count[palno][1] += 1;
3687         if (temp & 0x1)
3688             soft_drive_low_count[palno][0] += 1;
3689
3690         ++pin_spec_ptr;
3691     }
3692
3693     for (palno = 0; palno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++palno) {
3694         /* Check the HIGH current limit */
3695         total_current =
3696             PEL_DRIVE_HIGH_3_CURRENT * soft_drive_high_count[palno][3] +
3697             PEL_DRIVE_HIGH_2_CURRENT * soft_drive_high_count[palno][2] +
3698             PEL_DRIVE_HIGH_1_CURRENT * soft_drive_high_count[palno][1] +
3699             PEL_DRIVE_HIGH_0_CURRENT * soft_drive_high_count[palno][0];
3700
3701         /* Adjust the total high current depending on how far vab is from vcc */
3702         total_current += (((def_ptr->vcc - def_ptr->vab) * 4) / 2500 + 1);
3703
3704         if (total_current > MAX_PEL_DRIVE_CURRENT) {
3705             current_limit_exceeded = TRUE;
3706             in_queue_message(ERROR_MSG, "Pin Electronics Module in lane: %c slot: %d high current limit exceeded",
3707                             'A' + palno / MAX_SLOT_COUNT,
3708                             palno % MAX_SLOT_COUNT);
3709         }
3710     }
3711
3712     /* Check the LOW current limit */
3713     total_current =
3714         PEL_DRIVE_LOW_3_CURRENT * soft_drive_low_count[palno][3] +
3715         PEL_DRIVE_LOW_2_CURRENT * soft_drive_low_count[palno][2] +
3716         PEL_DRIVE_LOW_1_CURRENT * soft_drive_low_count[palno][1] +
3717         PEL_DRIVE_LOW_0_CURRENT * soft_drive_low_count[palno][0];
3718
3719
3720
3721

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE	5/23/89	PAGE #
TIME	6:14:53 pm	32/182

```

SOURCE TEXT
LINE #
3721 if (total_current > MAX_PEL_DRIVE_CURRENT) {
3722     current_limit_exceeded = TRUE;
3723     la_queue_message(ERROR MSG, "Pin Electronics Module is lane: tc slot: td low current limit exceeded",
3724         'A' + pelno / MAX_SLOT_COUNT,
3725         pelno % MAX_SLOT_COUNT);
3726 }
3727
3728 if (current_limit_exceeded == TRUE)
3729     return(FAILURE);
3730
3731 return(SUCCESS);
3732 }
3733
3734 program_dac(def_ptr, dab_info_index, changed_dac)
3735 DEVICE_SPEC *def_ptr;
3736 char dab_info_index;
3737 u_char *changed_dac;
3738 {
3739     EXTRA_DEVICE_SPEC *extra_def_ptr;
3740     DAB_INFO *dab_ptr;
3741     u_long addr;
3742     u_char unitno;
3743     u_char total_unit;
3744     u_char lane0;
3745     u_char slotno;
3746
3747     DPRINTF(("inside program_dac\n"));
3748
3749     *changed_dac = FALSE;
3750     extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;
3751     dab_ptr = dab_list[dab_info_index];
3752
3753     total_unit = dab_ptr->unit_count;
3754     for (unitno = 0; unitno < total_unit; ++unitno) {
3755         lane0 = dab_ptr->unit_location[unitno].lane_no;
3756         slotno = dab_ptr->unit_location[unitno].slot_no;
3757
3758         addr = LANE_0_START_ADDR + lane0 * LANE_ADDR_INC +
3759             LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC;
3760
3761         if (dab_ptr->val_programmed != extra_def_ptr->val_dac_value) {
3762             DPRINTF(("program VSL\n"));
3763             write_loc_long((u_long *))(addr + PEL_VSL_DAC_WRITE_OFFSET),
3764                 (u_long)extra_def_ptr->val_dac_value);
3765             *changed_dac = TRUE;
3766         }
3767
3768         if (dab_ptr->vsh_programmed != extra_def_ptr->vsh_dac_value) {
3769             DPRINTF(("program VSH\n"));
3770             write_loc_long((u_long *))(addr + PEL_VSH_DAC_WRITE_OFFSET),
3771                 (u_long)extra_def_ptr->vsh_dac_value);
3772             *changed_dac = TRUE;
3773         }
3774
3775         if (dab_ptr->vth_programmed != extra_def_ptr->vth_dac_value) {
3776             DPRINTF(("program VTH\n"));
3777             write_loc_long((u_long *))(addr + PEL_VTH_DAC_WRITE_OFFSET),
3778                 (u_long)extra_def_ptr->vth_dac_value);
3779             *changed_dac = TRUE;
3780         }
3781
3782         if (dab_ptr->vhl_programmed != extra_def_ptr->vhl_dac_value) {
3783             DPRINTF(("program VHL\n"));
3784             write_loc_long((u_long *))(addr + PEL_VHL_DAC_WRITE_OFFSET),
3785                 (u_long)extra_def_ptr->vhl_dac_value);
3786             *changed_dac = TRUE;
3787         }
3788
3789         if (dab_ptr->vil_programmed != extra_def_ptr->vil_dac_value) {
3790             DPRINTF(("program VIL\n"));
3791             write_loc_long((u_long *))(addr + PEL_VIL_DAC_WRITE_OFFSET),
3792                 (u_long)extra_def_ptr->vil_dac_value);
3793             *changed_dac = TRUE;
3794         }
3795
3796         if (dab_ptr->vih_programmed != extra_def_ptr->vih_dac_value) {
3797             DPRINTF(("program VIH\n"));
3798             write_loc_long((u_long *))(addr + PEL_VIH_DAC_WRITE_OFFSET),
3799                 (u_long)extra_def_ptr->vih_dac_value);
3800             *changed_dac = TRUE;
3801         }
3802
3803         /* If there are no DAC change on the first unit of the DAB,
3804            * there won't be any on other units, so just break from the
3805            * for loop.
3806            */
3807         if (*changed_dac == FALSE)
3808             break;
3809     }
3810
3811 #ifdef MODELER
3812     if (*changed_dac == TRUE) {
3813         DPRINTF(("changed dac value\n"));
3814         setup_timer1(DAC_TIMER_VALUE);
3815         dab_ptr->val_programmed = extra_def_ptr->val_dac_value;
3816         dab_ptr->vsh_programmed = extra_def_ptr->vsh_dac_value;
3817         dab_ptr->vth_programmed = extra_def_ptr->vth_dac_value;
3818         dab_ptr->vhl_programmed = extra_def_ptr->vhl_dac_value;
3819         dab_ptr->vil_programmed = extra_def_ptr->vil_dac_value;
3820         dab_ptr->vih_programmed = extra_def_ptr->vih_dac_value;
3821     }
3822 #endif
3823
3824     return(SUCCESS);
3825 }
3826
3827 set_private_mode(dab_ptr, set_it)
3828 DAB_INFO *dab_ptr;
3829 u_char set_it;
3830 {
3831     /* If "set_it" == TRUE then set the PRIVATE bit on each PEL
3832        * otherwise reset the bit.
3833        */
3834     u_long addr;
3835     u_char total_unit;
3836     u_char value;
3837     u_char lane0;
3838 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 33/183
LINE #		SOURCE TEXT		
3841		u_char slotno;		
3842		u_char unitno;		
3843				
3844		total_unit = dab_ptr->unit_count;		
3845		for (unitno = 0; unitno < total_unit; ++unitno) {		
3846		lane_no = dab_ptr->unit_location[unitno].lane_no;		
3847		slotno = dab_ptr->unit_location[unitno].slot_no;		
3848				
3849		addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +		
3850		LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +		
3851		PEL_STATUS_CONTROL_OFFSET;		
3852				
3853		value = read_loc_long((u_long *)addr) & 0xff;		
3854		if (set_it == TRUE) {		
3855		value  = ~PEL_CS_PRIVATE_PUBLIC_MASK;		
3856		}		
3857		else {		
3858		value  = PEL_CS_PRIVATE_PUBLIC_MASK;		
3859		}		
3860				
3861		write_loc_long((u_long *)addr, (u_long)value);		
3862		}		
3863				
3864				
3865		turn_on_is_use(dab_ptr)		
3866		DAB_INFO *dab_ptr;		
3867		{		
3868		u_long addr;		
3869		u_char total_unit;		
3870		u_char value;		
3871		u_char lane_no;		
3872		u_char slotno;		
3873		u_char unitno;		
3874				
3875		total_unit = dab_ptr->unit_count;		
3876		for (unitno = 0; unitno < total_unit; ++unitno) {		
3877		lane_no = dab_ptr->unit_location[unitno].lane_no;		
3878		slotno = dab_ptr->unit_location[unitno].slot_no;		
3879		}		
3880				
3881		addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +		
3882		LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +		
3883		PEL_STATUS_CONTROL_OFFSET;		
3884				
3885		value = read_loc_long((u_long *)addr) & 0xff   ~PEL_CS_IN_USE_LED_MASK;		
3886		write_loc_long((u_long *)addr, (u_long)value);		
3887		}		
3888				
3889				
3890		turn_off_is_use(dab_ptr)		
3891		DAB_INFO *dab_ptr;		
3892		{		
3893		u_long addr;		
3894		u_char total_unit;		
3895		u_char value;		
3896		u_char lane_no;		
3897		u_char slotno;		
3898		u_char unitno;		
3899				
3900		total_unit = dab_ptr->unit_count;		
3901		for (unitno = 0; unitno < total_unit; ++unitno) {		
3902		lane_no = dab_ptr->unit_location[unitno].lane_no;		
3903		slotno = dab_ptr->unit_location[unitno].slot_no;		
3904		}		
3905				
3906		addr = LANE_0_START_ADDR + lane_no * LANE_ADDR_INC +		
3907		LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +		
3908		PEL_STATUS_CONTROL_OFFSET;		
3909				
3910		value = read_loc_long((u_long *)addr) & 0xff & PEL_CS_IN_USE_LED_MASK;		
3911		write_loc_long((u_long *)addr, (u_long)value);		
3912		}		
3913				
3914				
3915		do_loop_ptr(instance)		
3916		INSTANCE_INFO *instance;		
3917		{		
3918		DEVICE_SPEC *def_ptr;		
3919		EXTRA_DEVICE_SPEC *extra_def_ptr;		
3920		u_long	inst_block_number[MAX_LANE_COUNT];	
3921		u_long	seq_end_addr[MAX_LANE_COUNT];	
3922		u_long	1;	
3923		u_long	timeout;	
3924		u_char	changed_dac;	
3925		extern int	lm_intr_requested; /* global interrupt advisory flag */	
3926				
3927		lm_intr_requested = 0;		
3928				
3929		/* The timeout value is set with the assumption that we are running at		
3930		* 150MHz (1/6us period).		
3931		* timeout = pattern_play_time * 1 second for feedback		
3932		*     = pattern_count * 8 microseconds + 125000 * 8 microseconds		
3933		*     = (pattern_count + 125000) * 8 microseconds		
3934		*     = (pattern_count + 125000) * 8 / 1000 millisecond		
3935		*     = (pattern_count + 125000) >> 7;		
3936		*     */		
3937		timeout = (instance->pattern_count + PTHM_COUNT_FUDGE_FACTOR) >> 7;		
3938				
3939		def_ptr = instance->definition;		
3940		extra_def_ptr = (EXTRA_DEVICE_SPEC *)def_ptr->extra_data;		
3941				
3942		if (program_dac(def_ptr, (char)instance->dab_info.index,		
3943		changed_dac) == FAILURE) {		
3944		return(FAILURE);		
3945		}		
3946				
3947		if (start_tmng(def_ptr) == FAILURE) {		
3948		return(FAILURE);		
3949		}		
3950		}		
3951		#endif		
3952				
3953		if (set_edge_and_sample_setting(extra_def_ptr,		
3954		(u_long)RESOLUTION_05_NS,		
3955		(u_long)MAX_SAMPLE_RANGE) == FAILURE)		
3956		return(FAILURE);		
3957				
3958		set_seq_end_bit(instance,		
3959		instance->lane_addr,		
3960		FALSE;		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
34/184

```

LINE #          SOURCE TEXT
3961          seq_end_addr,
3962          last_block_number);
3963
3964          for (i = 0; lm_instr_requested == 0; ++i) {
3965              if (play_ptrn_seq(instance, timeout, &changed_dac) == FAILURE) {
3966                  remove_seq_end_bit(instance, seq_end_addr, last_block_number);
3967                  if (i > 0)
3968                      lm_queue_message(ERROR_MSG, "failed in loop number: %d",
3969                                      i - lm_loop_pattern_count);
3970                  return(FAILURE);
3971              }
3972          }
3973
3974          remove_seq_end_bit(instance, seq_end_addr, last_block_number);
3975          return(SUCCESS);
3976      }
3977  }
3978
3979  count_avail_pattern(lanesum)
3980  u_char lanesum;
3981  {
3982      u_long total_block;
3983      u_long temp;
3984      u_char laneso;
3985
3986      if (lanesum == MAX_LANE_COUNT) {
3987          total_block = 0;
3988          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso)
3989              DPRINTF(("avail on lane: ALL -> %d blocks\n", total_block));
3990          return(total_block);
3991      }
3992      else {
3993          total_block = 0;
3994          temp = free_block_list[lanesum];
3995          while (temp != NULL) {
3996              ++total_block;
3997              temp = read_loc_long((u_long *)temp);
3998          }
3999          DPRINTF(("avail on lane: %d -> %d blocks\n", lanesum, total_block));
4000          return(total_block);
4001      }
4002  }
4003
4004  lm_free(ptr)
4005  char *ptr;
4006  {
4007      DPRINTF(("lm_free: %08X\n", ptr));
4008      free(ptr);
4009  }
4010
4011  char *
4012  lm_calloc(melem, elsize)
4013  unsigned melem, elsize;
4014  {
4015      char *ptr;
4016
4017      DPRINTF(("lm_calloc: %d elements of %d bytes -> ", melem, elsize));
4018      ptr = calloc(melem, elsize);
4019      DPRINTF((" %08X\n", ptr));
4020      return(ptr);
4021  }
4022
4023  char *
4024  lm_malloc(size)
4025  unsigned size;
4026  {
4027      char *ptr;
4028
4029      DPRINTF(("lm_malloc: %d bytes -> ", size));
4030      ptr = malloc(size);
4031      DPRINTF((" %08X\n", ptr));
4032      return(ptr);
4033  }
4034
4035  char *
4036  lm_realloc(ptr, size)
4037  char *ptr;
4038  unsigned size;
4039  {
4040      char *ptr2;
4041
4042      DPRINTF(("lm_realloc: ptr = %08X size = %08X -> ", ptr, size));
4043      ptr2 = realloc(ptr, size);
4044      DPRINTF((" %08X\n", ptr2));
4045      return(ptr2);
4046  }
4047
4048  get_fatal_hardware_message()
4049  {
4050      u_char laneso;
4051
4052      if (modeler_error.timing_error == TRUE)
4053          lm_queue_message(ERROR_MSG, "fatal Timing Generator error");
4054
4055      if (modeler_error.unknown_source_of_interrupt == TRUE) {
4056          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
4057              if (! (modeler_error.lane_errors & (1 << laneso)))
4058                  continue;
4059              lm_queue_message(ERROR_MSG, "unknown source of interrupt on lane: %c",
4060                              'A' + laneso);
4061          }
4062      }
4063
4064      if (modeler_error.lane_errors) {
4065          for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
4066              if (! (modeler_error.lane_errors & (1 << laneso)))
4067                  continue;
4068              if (modeler_error.pac_error[laneso].pac_refresh_error == TRUE)
4069                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller refresh error in lane: %c",
4070                              'A' + laneso);
4071              if (modeler_error.pac_error[laneso].pac_request_error == TRUE)
4072                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller request error in lane: %c",
4073                              'A' + laneso);
4074              if (modeler_error.pac_error[laneso].pac_pattern_error == TRUE)
4075                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller pattern error in lane: %c",
4076                              'A' + laneso);
4077              if (modeler_error.pac_error[laneso].
4078                  pac_control_word_parity_error == TRUE)
4079                  lm_queue_message(ERROR_MSG, "fatal Pattern Controller control word parity error in lane: %c",
4080                              'A' + laneso);
4081          }
4082      }
4083  }

```

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89

PAGE #

TIME 6:14:53 pm

35/185

```

LINE # SOURCE TEXT
4081 lm_queue_message(ERROR_MSG, "Fatal Pattern Controller CTL word parity error in lane: %c",
4082 'A' + lane_no);
4083 if (modeler_error.pac_error[lane_no])
4084     pac_high_word_parity_error == TRUE)
4085 lm_queue_message(ERROR_MSG, "Fatal Pattern Controller high word parity error in lane: %c",
4086 'A' + lane_no);
4087 if (modeler_error.pac_error[lane_no])
4088     pac_low_word_parity_error == TRUE)
4089 lm_queue_message(ERROR_MSG, "Fatal Pattern Controller low word parity error in lane: %c",
4090 'A' + lane_no);
4091 }
4092 }
4093 }
4094 }
4095 check_pel_errors(instance);
4096 INSTANCE_INFO "instance:
4097 {
4098     DAB_INFO      *dab_ptr;
4099     pel_error_list
4100     u_long      mask;
4101     u_char      cur_pelno;
4102     u_char      pelno;
4103     u_char      magicno;
4104     u_char      unitno;
4105     u_char      magic_mask;
4106     u_char      found_error;
4107     u_char      error_on_this_instance = FALSE;
4108 }
4109 dab_ptr = dab_list(instance->dab_info_index);
4110
4111 pel_error_list = modeler_error.pel_error_list;
4112 modeler_error.pel_error_list = 0;
4113 for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {
4114     if (pel_error_list == 0)
4115         break;
4116     found_error = FALSE;
4117     mask = 1 << pelno;
4118     if (! (pel_error_list & mask))
4119         continue;
4120     if (system_config->lane[pelno / MAX_SLOT_COUNT] == NULL)
4121         continue;
4122     pel_error_list ^= mask;
4123     /* Check PLAY error */
4124     if (modeler_error.pel_error[pelno].play_error == TRUE) {
4125         lm_queue_message(ERROR_MSG, "play error in lane: %c slot: %d",
4126             'A' + pelno / MAX_SLOT_COUNT,
4127             pelno % MAX_SLOT_COUNT);
4128         found_error = TRUE;
4129         modeler_error.pel_error[pelno].play_error = FALSE;
4130     }
4131     /* Check MAGIC PARITY error */
4132     if (modeler_error.pel_error[pelno].any_magic_parity == TRUE) {
4133         found_error = TRUE;
4134         modeler_error.pel_error[pelno].any_magic_parity = FALSE;
4135         for (magicno = 0; magicno < MAX_MAGIC_COUNT; ++magicno) {
4136             magic_mask = 1 << magicno;
4137             if ((modeler_error.pel_error[pelno].
4138                 magic_parity_out & magic_mask) != 0) {
4139                 lm_queue_message(ERROR_MSG, "Pin Electronics Module parity error in lane: %c slot: %d channel: %d",
4140                     'A' + pelno / MAX_SLOT_COUNT,
4141                     pelno % MAX_SLOT_COUNT,
4142                     magicno);
4143                 modeler_error.pel_error[pelno].magic_parity_out ^= magic_mask;
4144             }
4145         }
4146     }
4147     /* Check MAGIC SHORT error */
4148     if (modeler_error.pel_error[pelno].any_magic_short == TRUE) {
4149         /* Find the unitno of this pel */
4150         for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4151             cur_pelno =
4152                 dab_ptr->unit_location[unitno].lane_no * 8 +
4153                 dab_ptr->unit_location[unitno].slot_no;
4154             if (cur_pelno == pelno) {
4155                 break;
4156             }
4157         }
4158         if (unitno == dab_ptr->unit_count) {
4159             mark_instance_shorted_pin(pelno);
4160             init_dab(pelno / MAX_SLOT_COUNT, pelno % MAX_SLOT_COUNT);
4161             continue;
4162         }
4163         found_error = TRUE;
4164         report_shorted_pin(instance);
4165     }
4166     if (found_error == TRUE) {
4167         error_on_this_instance = TRUE;
4168         init_dab(pelno / MAX_SLOT_COUNT, pelno % MAX_SLOT_COUNT);
4169     }
4170 }
4171 if (error_on_this_instance == TRUE)
4172     return(FAILURE);
4173 return(SUCCESS);
4174 }
4175
4176 mark_instance_shorted_pin(pelno)
4177 u_char pelno;
4178 {
4179     USER_INFO      *user;
4180     INSTANCE_INFO   *instance;
4181     DAB_INFO         *dab_ptr;
4182     u_short          inst_id;
4183     u_char           username;
4184 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89 PAGE #  
TIME 6:14:53 pm 36/186

```

LINE # SOURCE TEXT
4201 u_char          unitno;
4202 u_char          cur_pelso;
4203
4204 for (userno = 0; userno < MAX_USER_COUNT; ++userno) {
4205     user = user_info_array[userno];
4206     if (user->active == FALSE)
4207         continue;
4208     for (inst_id = 0; inst_id < user->inst_table_size; ++inst_id) {
4209         instance = user->instance[inst_id];
4210         if (ISOCES_INSTANCE(user, instance))
4211             continue;
4212         dab_ptr = dab_list[instance->dab_info_index];
4213         for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4214             cur_pelso =
4215                 dab_ptr->unit_location[unitno].lane_no * 8 +
4216                 dab_ptr->unit_location[unitno].slot_no;
4217             if (cur_pelso == pelso) {
4218                 /* This instance is using the pelso */
4219                 instance->had_shorted_pin = TRUE;
4220                 return;
4221             }
4222         }
4223     }
4224 }
4225
4226 report_shorted_pin(instance)
4227 INSTANCE_INFO *instance;
4228 {
4229     DEVICE_SPEC *definition;
4230     DAB_INFO *dab_ptr;
4231     u_short magic_short;
4232     u_short pin_number;
4233     u_short bit_mask;
4234     u_char unitno;
4235     u_char pelso;
4236     u_char magicno;
4237     u_char bitno;
4238     u_char unit_pin_number;
4239     u_char direction;
4240
4241     dab_ptr = dab_list[instance->dab_info_index];
4242     definition = instance->definition;
4243     for (unitno = 0; unitno < dab_ptr->unit_count; ++unitno) {
4244         pelso =
4245             dab_ptr->unit_location[unitno].lane_no * 8 +
4246             dab_ptr->unit_location[unitno].slot_no;
4247         if (modeler_error.pel_error[pelso].any_magic_short == TRUE) {
4248             modeler_error.pel_error[pelso].any_magic_short = FALSE;
4249             for (magicno = 0; magicno < MAX_MAGIC_COUNT; ++magicno) {
4250                 magic_short = modeler_error.pel_error[pelso].magic_short[magicno];
4251                 modeler_error.pel_error[pelso].magic_short[magicno] = 0;
4252                 if (magic_short != 0) {
4253                     for (bitno = 0; bitno < 16; ++bitno) {
4254                         if (magic_short == 0)
4255                             break;
4256                         bit_mask = 1 << bitno;
4257                         if (! (magic_short & bit_mask))
4258                             continue;
4259                         magic_short ^= bit_mask;
4260                         unit_pin_number = magicno * 16 + bitno;
4261                         pin_number = unitno * 80 + unit_pin_number;
4262                         direction = definition->pin_table[pin_number].direction;
4263                         if ((direction == NONE) ||
4264                             (direction == POWER) ||
4265                             (direction == GROUND) ||
4266                             (direction == NC))
4267                             continue;
4268                         in_queue_message(ERROR_MSG, "internal error: shorted NONE/POWER/GROUND/NC pin in lane: %c slot: %d unit PN: %d",
4269                                         'A' + pelso / MAX_SLOT_COUNT,
4270                                         pelso % MAX_SLOT_COUNT,
4271                                         unit_pin_number);
4272                     }
4273                     else
4274                         in_queue_message(ERROR_MSG, "pin: %s of instance: %s is shorted",
4275                                         definition->pin_table[pin_number].pin_name,
4276                                         instance->device_info_string);
4277                 }
4278             }
4279         }
4280     }
4281 }
4282
4283 init_err()
4284 {
4285     u_long junk;
4286     u_char laneso;
4287     u_char pelso;
4288     u_char magicno;
4289
4290     if (read_tag(&junk) == SUCCESS) {
4291         tagptr->lane_intr_enable = 1;
4292         tagptr->tag_intr_enable = 1;
4293         tagptr->tag_intr_clear1 = 1;
4294     }
4295
4296     modeler_error.pel_error_list = 0;
4297     modeler_error.error = FALSE;
4298     modeler_error.lane_errors = 0;
4299     modeler_error.pac_lane_errors = 0;
4300     modeler_error.dab_change = FALSE;
4301     modeler_error.tag_error = FALSE;
4302
4303     for (laneso = 0; laneso < MAX_LANE_COUNT; ++laneso) {
4304         modeler_error.pac_error[laneso].pac_refresh_error = FALSE;
4305     }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89	PAGE # 37/187
TIME 6:14:53 pm				
LINE #	SOURCE TEXT			
4321	modeler_error.pac_error(lasemo).pac_request_error = FALSE;			
4322	modeler_error.pac_error(lasemo).pac_patterns_error = FALSE;			
4323	modeler_error.pac_error(lasemo).pac_control_word_parity_error = FALSE;			
4324	modeler_error.pac_error(lasemo).pac_high_word_parity_error = FALSE;			
4325	modeler_error.pac_error(lasemo).pac_low_word_parity_error = FALSE;			
4326	modeler_error.pac_error(lasemo).pac_branch_address = 0;			
4327	modeler_error.pac_error(lasemo).pac_block_offset = 0;			
4328	modeler_error.pac_error(lasemo).pac_parity_error_address = 0;			
4329	}			
4330	for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {			
4331	modeler_error.pel_error(pelno).dab_inserted = FALSE;			
4332	modeler_error.pel_error(pelno).dab_removed = FALSE;			
4333	modeler_error.pel_error(pelno).play_error = FALSE;			
4334	modeler_error.pel_error(pelno).asy_magic_short = FALSE;			
4335	modeler_error.pel_error(pelno).asy_magic_parity = FALSE;			
4336	modeler_error.pel_error(pelno).magic_parity_out = 0;			
4337	}			
4338	for (magicno = 0; magicno < 5; ++magicno) {			
4339	modeler_error.pel_error(pelno).magic_short[magicno] = 0;			
4340	}			
4341	}			
4342	}			
4343	}			
4344	}			
4345	clear_non_fatal_mod_err()			
4346	{			
4347	u_char pelno;			
4348	u_char magicno;			
4349	modeler_error.pel_error_list = 0;			
4350	for (pelno = 0; pelno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++pelno) {			
4351	modeler_error.pel_error(pelno).dab_inserted = FALSE;			
4352	modeler_error.pel_error(pelno).dab_removed = FALSE;			
4353	modeler_error.pel_error(pelno).play_error = FALSE;			
4354	modeler_error.pel_error(pelno).asy_magic_short = FALSE;			
4355	modeler_error.pel_error(pelno).asy_magic_parity = FALSE;			
4356	modeler_error.pel_error(pelno).magic_parity_out = 0;			
4357	}			
4358	for (magicno = 0; magicno < 5; ++magicno) {			
4359	modeler_error.pel_error(pelno).magic_short[magicno] = 0;			
4360	}			
4361	}			
4362	}			
4363	}			
4364	}			
4365	lm_config_error(error_type, lasemo, slotno)			
4366	u_char error_type;			
4367	u_char lasemo;			
4368	u_char slotno;			
4369	{			
4370	switch (error_type) {			
4371	case CERR_TWG_CAL:			
4372	config_error.twg_cal = 1;			
4373	fatal_configuration_error_encountered = TRUE;			
4374	break;			
4375	case CERR_NO_TWG:			
4376	config_error.no_twg = 1;			
4377	fatal_configuration_error_encountered = TRUE;			
4378	break;			
4379	case CERR_NO_PAM_FOR_PAC:			
4380	config_error.no_pam_for_pac  = 1 << lasemo;			
4381	fatal_configuration_error_encountered = TRUE;			
4382	break;			
4383	case CERR_PAM_STRAPPED_WRONG:			
4384	config_error.pam_strapped_wrong  =			
4385	1 << (lasemo * MAX_PAM_COUNT + slotno);			
4386	fatal_configuration_error_encountered = TRUE;			
4387	break;			
4388	case CERR_PAM_STACKED_WRONG:			
4389	config_error.pam_stacked_wrong  =			
4390	1 << (lasemo * MAX_PAM_COUNT + slotno);			
4391	fatal_configuration_error_encountered = TRUE;			
4392	break;			
4393	case CERR_DUPLICATE_SEGMENT:			
4394	config_error.duplicate_segment  =			
4395	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4396	non_fatal_configuration_error_encountered = TRUE;			
4397	break;			
4398	case CERR_MISSING_SEGMENT:			
4399	config_error.missing_segment  =			
4400	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4401	non_fatal_configuration_error_encountered = TRUE;			
4402	break;			
4403	case CERR_NO_PEL_FOR_DAB:			
4404	config_error.no_pel_for_dab  =			
4405	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4406	non_fatal_configuration_error_encountered = TRUE;			
4407	break;			
4408	case CERR_NO_PAM_FOR_DAB:			
4409	config_error.no_pam_for_dab  =			
4410	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4411	non_fatal_configuration_error_encountered = TRUE;			
4412	break;			
4413	case CERR_ILLEGAL_DUPLICATE_DEVICE:			
4414	config_error.illegal_duplicate_device  =			
4415	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4416	non_fatal_configuration_error_encountered = TRUE;			
4417	break;			
4418	case CERR_DEVICE_TOO_LARGE:			
4419	config_error.device_too_large  =			
4420	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4421	non_fatal_configuration_error_encountered = TRUE;			
4422	break;			
4423	case CERR_ILLEGAL_PEL_STACKING:			
4424	config_error.illegal_pel_stacking  =			
4425	1 << (lasemo * MAX_SLOT_COUNT + slotno);			
4426	non_fatal_configuration_error_encountered = TRUE;			
4427	break;			
4428	default:			
4429	DPRINTF(("illegal error_type is config_error\n"));			
4430	break;			
4431	}			
4432	}			
4433	}			
4434	init_config_error()			
4435	{			
4436	u_long error;			
4437	config_error.do_twg_calibration = 0;			
4438	}			
4439	}			
4440	#ifdef MODELER			



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89 TIME 6:14:53 pm	PAGE # 38/188
LINE #	SOURCE TEXT			
4441	(void)lm_svaram_access((char *)&config_error, CONFIG_ERROR,			
4442	(u_long)sizeof(CONFIGURATION_ERRORS), MEMORY_READ,			
4443	&error);			
4444	#endif			
4445	config_error.tmg_cal = 0;			
4446	config_error.no_tmg = 0;			
4447	config_error.no_pam_for_pac = 0;			
4448	config_error.pam_strapped_wrong = 0;			
4449	config_error.pam_stacked_wrong = 0;			
4450	config_error.duplicate_segment = 0;			
4451	config_error.missing_segment = 0;			
4452	config_error.no_pai_for_dab = 0;			
4453	config_error.no_pam_for_dab = 0;			
4454	config_error.illegal_duplicate_device = 0;			
4455	config_error.device_too_large = 0;			
4456	config_error.illegal_pai_stacking = 0;			
4457	}			
4458	}			
4459	write_config_error();			
4460	{			
4461	#ifdef MODELER			
4462	u_long error;			
4463	DPRINTF(("write config_error: CONFIG_ERROR: %08x, size: %d\n",			
4464	CONFIG_ERROR, sizeof(CONFIGURATION_ERRORS)));			
4465	DPRINTF(("tmg_cal = %d\n",			
4466	config_error.tmg_cal));			
4467	DPRINTF(("no_tmg = %d\n",			
4468	config_error.no_tmg));			
4469	DPRINTF(("no_pam_for_pac = %08x\n",			
4470	config_error.no_pam_for_pac));			
4471	DPRINTF(("pam_strapped_wrong = %08x\n",			
4472	config_error.pam_strapped_wrong));			
4473	DPRINTF(("pam_stacked_wrong = %08x\n",			
4474	config_error.pam_stacked_wrong));			
4475	DPRINTF(("duplicate_segment = %08x\n",			
4476	config_error.duplicate_segment));			
4477	DPRINTF(("missing_segment = %08x\n",			
4478	config_error.missing_segment));			
4479	DPRINTF(("no_pai_for_dab = %08x\n",			
4480	config_error.no_pai_for_dab));			
4481	DPRINTF(("no_pam_for_dab = %08x\n",			
4482	config_error.no_pam_for_dab));			
4483	DPRINTF(("illegal_duplicate_device = %08x\n",			
4484	config_error.illegal_duplicate_device));			
4485	DPRINTF(("device_too_large = %08x\n",			
4486	config_error.device_too_large));			
4487	DPRINTF(("illegal_pai_stacking = %08x\n",			
4488	config_error.illegal_pai_stacking));			
4489	(void)lm_svaram_access((char *)&config_error, CONFIG_ERROR,			
4490	(u_long)sizeof(CONFIGURATION_ERRORS), MEMORY_WRITE,			
4491	&error);			
4492	#endif			
4493	}			
4494	write_user_stat(patterns_count, patterns_count1, longest_patterns_seq,			
4495	def_count, inst_count, fault_count);			
4496	u_long patterns_count;			
4497	u_long patterns_count1;			
4498	u_long longest_patterns_seq;			
4499	u_short def_count;			
4500	u_short inst_count;			
4501	u_short fault_count;			
4502	#ifdef MODELER			
4503	u_long error;			
4504	#endif			
4505	RUNTIME_STAT_STRUCT svaram_value;			
4506	#ifdef MODELER			
4507	(void)lm_svaram_access((char *)&svaram_value, RUNTIME_STAT,			
4508	(u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_READ,			
4509	&error);			
4510	#endif			
4511	add_64(&svaram_value.patterns_count1,			
4512	&svaram_value.patterns_count1,			
4513	&patterns_count, patterns_count1);			
4514	if (&svaram_value.longest_patterns_seq < longest_patterns_seq)			
4515	&svaram_value.longest_patterns_seq = longest_patterns_seq;			
4516	&svaram_value.definition_count += def_count;			
4517	&svaram_value.instance_count += inst_count;			
4518	&svaram_value.fault_count += fault_count;			
4519	#ifdef MODELER			
4520	(void)lm_svaram_access((char *)&svaram_value, RUNTIME_STAT,			
4521	(u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_WRITE,			
4522	&error);			
4523	#endif			
4524	add_64(op1h, op1l, op2h, op2l)			
4525	u_long op1h;			
4526	u_long op1l;			
4527	u_long op2h;			
4528	u_long op2l;			
4529	{			
4530	/* 64 bit add: *op1 <- *op1 + *op2 */			
4531	u_long th;			
4532	u_long tl;			
4533	th = *op1h + *op2h;			
4534	tl = *op1l + *op2l;			
4535	if (((*op1h & 0x80000000) & (*op2h & 0x80000000))			
4536	((( *op1l & 0x80000000)    (*op2l & 0x80000000)) & (*tl & 0x80000000)))			
4537	++th;			
4538	*op1h = th;			
4539	*op1l = tl;			
4540	}			
4541	#endif			
4542	}			
4543	}			
4544	}			
4545	}			
4546	}			
4547	}			
4548	}			
4549	}			
4550	}			
4551	}			
4552	}			
4553	}			
4554	}			
4555	}			
4556	}			
4557	}			
4558	}			
4559	}			
4560	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
lm1000/util.c

DATE 5/23/89  
TIME 6:14:53 pm

PAGE #  
39/189

```

LINE # SOURCE TEXT
4561 update_svarm_dab_insertion(loc_dab_list)
4562 {
4563     DAB_INFO *loc_dab_list;
4564     DAB_INFO *dab_ptr;
4565     #ifdef MODELER
4566     .b esprun;
4567     u_long svarm_dab_insertion;
4568     u_long error;
4569     #endif
4570     u_char dab_insertion = 0;
4571     u_char dabno;
4572     char segno;
4573
4574     for (dabno = 0; dabno < MAX_LANE_COUNT * MAX_SLOT_COUNT; ++dabno) {
4575         if (loc_dab_list[dabno] == NULL)
4576             continue;
4577         dab_ptr = loc_dab_list[dabno];
4578
4579         for (segno = 0; segno <= MAX_SEGMENT_PER_DEVICE; ++segno) {
4580             if (dab_ptr->segment(segno) == NULL)
4581                 continue;
4582
4583             if (dab_active(dab_ptr->segment(segno)->lane_no,
4584                 dab_ptr->segment(segno)->slot_no) == FALSE) {
4585                 ++dab_insertion;
4586             }
4587         }
4588     }
4589     #ifdef MODELER
4590     (void)lm_write_esprun_in_count(
4591         dab_ptr->segment(segno)->lane_no * MAX_SLOT_COUNT +
4592         dab_ptr->segment(segno)->slot_no,
4593         (u_short)dab_ptr->segment(segno)->insertion_count + 1);
4594     /* ??? */
4595     (void)lm_read_esprun(
4596         dab_ptr->segment(segno)->lane_no * MAX_SLOT_COUNT +
4597         dab_ptr->segment(segno)->slot_no,
4598         &dab_esprun);
4599
4600     if (dab_esprun.insertion_count !=
4601         dab_ptr->segment(segno)->insertion_count + 1) {
4602         DPRINTF(("DAB EEPROM write insertion_count failed\n"));
4603         DPRINTF(("wrote: %d read: %d\n",
4604             dab_ptr->segment(segno)->insertion_count + 1,
4605             dab_esprun.insertion_count));
4606     }
4607     #endif
4608 }
4609
4610 #endif
4611
4612 #ifdef MODELER
4613 (void)lm_svarm_access((char *)&svarm_dab_insertion, RUNTIME_STAT,
4614     (u_long)sizeof(u_long), MEMORY_READ,
4615     &error);
4616 svarm_dab_insertion += dab_insertion;
4617 (void)lm_svarm_access((char *)&svarm_dab_insertion, RUNTIME_STAT,
4618     (u_long)sizeof(u_long), MEMORY_WRITE,
4619     &error);
4620 #endif
4621
4622 dab_active(laneno, slotno)
4623 u_char laneno;
4624 u_char slotno;
4625 {
4626     u_long addr;
4627     u_short value;
4628
4629     addr = LANE_0_START_ADDR + laneno * LANE_ADDR_INC +
4630         LANE_PEL_0_START_OFFSET + slotno * LANE_PEL_ADDR_INC +
4631         PEL_STATUS_CONTROL_OFFSET;
4632     value = (u_short)read_loc_long((u_long *)&addr);
4633
4634     if (((value & PEL_CS_ACTIVE_MASK) == 0) ||
4635         ((value & PEL_CS_INITIALIZE_MASK) == 0) ||
4636         ((value & PEL_CS_RESET_MASK) == 0))
4637         return(FALSE);
4638     return(TRUE);
4639 }
4640
4641 fatal_alloc_error()
4642 {
4643     /* ??? */
4644 }
4645
4646 add_inconsistent_pins(instance, ident_inconsistent_pins)
4647 INSTANCE_INFO *instance;
4648 PTRN_BITS_LONGWORD *ident_inconsistent_pins;
4649 {
4650     DAB_INFO *dab_ptr;
4651     u_long *unknown_ptr;
4652     u_char total_word;
4653     u_char wordno;
4654
4655     dab_ptr = dab_list[instance->dab_info_index];
4656     total_word = dab_ptr->unit_count *
4657         sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);
4658     unknown_ptr = (u_long *)instance->last_sample_value.unknown;
4659     for (wordno = 0; wordno < total_word; ++wordno) {
4660         unknown_ptr[wordno] |= ((u_long *)ident_inconsistent_pins)[wordno];
4661     }
4662 }
4663
4664 epoch_info(attr, value)
4665 u_short attr;
4666 u_long *value;
4667 {
4668     #ifdef MODELER
4669     u_long error;
4670     RUNTIME_STAT_STRUCT rstat;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89 TIME 6:14:53 pm	PAGE # 40/190
LINE #	SOURCE TEXT			
4681	(void)lm_sparam_access((char *)rstat, RUNTIME_STAT,			
4682	{(u_long)sizeof(RUNTIME_STAT_STRUCT), MEMORY_READ,			
4683	error);			
4684				
4685				
4686	switch (attr) {			
4687	case LM_EPOCH_ADAPTER_INSERTIONS:			
4688	value = rstat.dab_insertion_count;			
4689	break;			
4690	case LM_EPOCH_LONGEST_SEQUENCE:			
4691	value = rstat.longest_pattern_seq;			
4692	break;			
4693	case LM_EPOCH_PATTERNS_ADDED_HIGH:			
4694	value = rstat.pattern_count_high;			
4695	break;			
4696	case LM_EPOCH_PATTERNS_ADDED_LOW:			
4697	value = rstat.pattern_count_low;			
4698	break;			
4699	case LM_EPOCH_DEFINITIONS:			
4700	value = rstat.definition_count;			
4701	break;			
4702	case LM_EPOCH_INSTANCES:			
4703	value = rstat.instance_count;			
4704	break;			
4705	case LM_EPOCH_FAULTS:			
4706	value = rstat.fault_count;			
4707	break;			
4708	default:			
4709	return (FAILURE);			
4710	}			
4711	{			
4712	switch (attr) {			
4713	case LM_EPOCH_ADAPTER_INSERTIONS:			
4714	case LM_EPOCH_LONGEST_SEQUENCE:			
4715	case LM_EPOCH_PATTERNS_ADDED_HIGH:			
4716	case LM_EPOCH_PATTERNS_ADDED_LOW:			
4717	case LM_EPOCH_DEFINITIONS:			
4718	case LM_EPOCH_INSTANCES:			
4719	case LM_EPOCH_FAULTS:			
4720	value = 0;			
4721	break;			
4722	default:			
4723	return (FAILURE);			
4724	}			
4725	{			
4726	return (SUCCESS);			
4727	}			
4728				
4729	adjust_delay(user, def_ptr)			
4730	USER_INFO user;			
4731	DEVICE_SPEC def_ptr;			
4732	{			
4733	u_short i;			
4734	u_short mtyp_count;			
4735				
4736	/* Adjust the default delay if specified, otherwise set it to 1 */			
4737	if (def_ptr->use_default == TRUE) {			
4738	ADJUST_DELAY(def_ptr->default_delay.minimum,			
4739	user->time_scale,			
4740	user->half_time_scale,			
4741	user->divide_delay);			
4742	ADJUST_DELAY(def_ptr->default_delay.typical,			
4743	user->time_scale,			
4744	user->half_time_scale,			
4745	user->divide_delay);			
4746	ADJUST_DELAY(def_ptr->default_delay.maximum,			
4747	user->time_scale,			
4748	user->half_time_scale,			
4749	user->divide_delay);			
4750	}			
4751	{			
4752	def_ptr->default_delay.minimum = 1;			
4753	def_ptr->default_delay.typical = 1;			
4754	def_ptr->default_delay.maximum = 1;			
4755	}			
4756				
4757	mtyp_count = def_ptr->mtyp_cnt;			
4758	for (i = 0; i < mtyp_count; ++i) {			
4759	if ((long)def_ptr->mtyp_table[i].minimum != -1)			
4760	ADJUST_DELAY(def_ptr->mtyp_table[i].minimum,			
4761	user->time_scale,			
4762	user->half_time_scale,			
4763	user->divide_delay);			
4764				
4765	if ((long)def_ptr->mtyp_table[i].typical != -1)			
4766	ADJUST_DELAY(def_ptr->mtyp_table[i].typical,			
4767	user->time_scale,			
4768	user->half_time_scale,			
4769	user->divide_delay);			
4770	if ((long)def_ptr->mtyp_table[i].maximum != -1)			
4771	ADJUST_DELAY(def_ptr->mtyp_table[i].maximum,			
4772	user->time_scale,			
4773	user->half_time_scale,			
4774	user->divide_delay);			
4775	}			
4776				
4777				
4778	unadjust_delay(user, def_ptr)			
4779	USER_INFO user;			
4780	DEVICE_SPEC def_ptr;			
4781	{			
4782	u_short i;			
4783	u_short mtyp_count;			
4784				
4785	/* Unadjust the default delay if specified */			
4786	if (def_ptr->use_default == TRUE) {			
4787	UNADJUST_DELAY(def_ptr->default_delay.minimum,			
4788	user->time_scale,			
4789	user->half_time_scale,			
4790	user->divide_delay);			
4791	UNADJUST_DELAY(def_ptr->default_delay.typical,			
4792	user->time_scale,			
4793	user->half_time_scale,			
4794	user->divide_delay);			
4795	UNADJUST_DELAY(def_ptr->default_delay.maximum,			
4796	user->time_scale,			
4797	user->half_time_scale,			
4798	user->divide_delay);			
4799	}			
4800				

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89 TIME 6:14:53 pm	PAGE # 41/191
LINE #	SOURCE TEXT			
4801	mtye count = def_ptr->mtye_cnt;			
4802	for (i = 0; i < mtye_count; ++i) {			
4803	if ((long)def_ptr->mtye_table[i].minimum != -1)			
4804	UNADJUST_DELAY(def_ptr->mtye_table[i].minimum,			
4805	user->time_scale,			
4806	user->half_time_scale,			
4807	user->divide_delay);			
4808	if ((long)def_ptr->mtye_table[i].typical != -1)			
4809	UNADJUST_DELAY(def_ptr->mtye_table[i].typical,			
4810	user->time_scale,			
4811	user->half_time_scale,			
4812	user->divide_delay);			
4813	if ((long)def_ptr->mtye_table[i].maximum != -1)			
4814	UNADJUST_DELAY(def_ptr->mtye_table[i].maximum,			
4815	user->time_scale,			
4816	user->half_time_scale,			
4817	user->divide_delay);			
4818	}			
4819	}			
4820	}			
4821	}			
4822	}			
4823	set_time_scale(user, number, units)			
4824	USER_INFO *user;			
4825	u_long number;			
4826	long units;			
4827	{			
4828	double temp;			
4829	switch (units) {			
4830	case LM_FEMTOSECONDS:			
4831	temp = 1000.0;			
4832	break;			
4833	case LM_PICOSECONDS:			
4834	temp = 1.0;			
4835	break;			
4836	case LM_NANOSECONDS:			
4837	temp = 0.001;			
4838	break;			
4839	case LM_MICROSECONDS:			
4840	temp = 0.000001;			
4841	break;			
4842	default:			
4843	lm_queue_message(ERROR_MSG, "internal simulator error: illegal unit: %d for simulation tick",			
4844	units);			
4845	return(Failure);			
4846	}			
4847	temp = temp / (double)number;			
4848	if (temp > 1.0) {			
4849	user->divide_delay = FALSE;			
4850	user->time_scale = (u_long)temp;			
4851	}			
4852	else {			
4853	user->divide_delay = TRUE;			
4854	user->time_scale = (u_long)(1.0 / temp);			
4855	}			
4856	user->half_time_scale = user->time_scale / 2;			
4857	/* Just in case */			
4858	if (user->time_scale == 0)			
4859	user->time_scale = 1;			
4860	return(SUCCESS);			
4861	}			
4862	}			
4863	mask_off_output(instance, setup_ptr)			
4864	INSTANCE_INFO *instance;			
4865	PTRN_BITS *setup_ptr;			
4866	{			
4867	/* Mask off the ptrn bits for outputs in the measurement pattern.			
4868	* This has to be done because for output pins we only have the soft			
4869	* drive 0 turned on, so we need to pull the outputs low.			
4870	*/			
4871	EXTRA_DEVICE_SPEC *extra_def_ptr;			
4872	DAB_INFO *dab_ptr;			
4873	u_long *setup_ptr_ptr;			
4874	u_long *ident_outputs_ptr;			
4875	u_char total_word;			
4876	u_char wordao;			
4877	extra_def_ptr = (EXTRA_DEVICE_SPEC *)instance->definition->extra_data;			
4878	dab_ptr = dab_list(instance->dab_info_index);			
4879	total_word = dab_ptr->unit_count;			
4880	sizeof(PTRN_BITS_LONGWORD) / sizeof(u_long);			
4881	setup_ptr_ptr = (u_long *)setup_ptr;			
4882	ident_outputs_ptr = (u_long *)extra_def_ptr->ident_outputs;			
4883	for (wordao = 0; wordao < total_word; ++wordao) {			
4884	*setup_ptr_ptr += *ident_outputs_ptr;			
4885	++setup_ptr_ptr;			
4886	++ident_outputs_ptr;			
4887	}			
4888	}			
4889	}			
4890	lm_crypt(password, crypted_string)			
4891	char *password;			
4892	char *crypted_string;			
4893	/* If you change this you MUST change the backdoor password */			
4894	{			
4895	char *temp_ptr;			
4896	char password_buffer(CRYPTED_PASSWORD_LENGTH);			
4897	int i;			
4898	char c;			
4899	DPRINTF(("inside lm_crypt; password: %s\n", password));			
4900	if (strlen(password) == 0) {			
4901	for (i = 0; i < CRYPTED_PASSWORD_LENGTH; ++i) {			
4902	crypted_string[i] = '\0';			
4903	}			
4904	return(SUCCESS);			
4905	}			
4906	}			
4907	i = 0;			
4908	}			
4909	}			
4910	}			
4911	}			
4912	}			
4913	}			
4914	}			
4915	}			
4916	}			
4917	}			
4918	}			
4919	}			
4920	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM lm1000/util.c	DATE 5/23/89 TIME 6:14:53 pm	PAGE # 42/192
LINE #	SOURCE TEXT			
4921	while (1 < CRYPTED_PASSWORD_LENGTH) {			
4922	temp_ptr = password;			
4923	while ((c = *temp_ptr++) && (1 < CRYPTED_PASSWORD_LENGTH)) {			
4924	password_buffer[i++] = c;			
4925	}			
4926	}			
4927	for (i=0; i<CRYPTED_PASSWORD_LENGTH; i++)			
4928	crypt_string[i] =			
4929	(((char) 1 + strlen(password) << 3) ^ (password_buffer[i] ^ (password_buffer[CRYPTED_PASSWORD_LENGTH-(i+1)] << 1)));			
4930	}			
4931	return(SUCCESS);			
4932	}			
4933	}			
4934	}			
4935	check_password(user)			
4936	USER_INFO *user;			
4937	{			
4938	char    good_password[CRYPTED_PASSWORD_LENGTH];			
4939	u_long  status;			
4940	u_char  i;			
4941	}			
4942	DPRINTF(("inside check_password\n"));			
4943	if (user->good_password == TRUE) {			
4944	DPRINTF(("user->good_password is TRUE\n"));			
4945	return(SUCCESS);			
4946	}			
4947	if (proc_lm_rd(LM_NVRAM_MEMORY, 0, PASSWORD,			
4948	sizeof PASSWORD, good_password, &status) == FAILURE) {			
4949	lm_queue_message(ERROR_MSG, "internal error: Failed to read password");			
4950	DPRINTF(("error in read\n"));			
4951	return(FAILURE);			
4952	}			
4953	/* Check for no password */			
4954	for (i = 0; i < CRYPTED_PASSWORD_LENGTH; ++i) {			
4955	if (good_password[i] != '\0')			
4956	break;			
4957	}			
4958	if (i == CRYPTED_PASSWORD_LENGTH) {			
4959	DPRINTF(("No password is set\n"));			
4960	lm_queue_message(WARNING_MSG, "No password set on modeler");			
4961	return(SUCCESS);			
4962	}			
4963	/* Check for CPU switch 3 */			
4964	status = read_loc_long((u_long *)CPU_STATUS_REG_ADDR);			
4965	if (status & CPU_BIP_SWITCH_MASK) {			
4966	DPRINTF(("Password not checked due to hardware override\n"));			
4967	lm_queue_message(WARNING_MSG, "Password not checked due to hardware override");			
4968	return(SUCCESS);			
4969	}			
4970	DPRINTF(("error in check_password\n"));			
4971	return(FAILURE);			
4972	}			
4973	}			
4974	write_password(encrypted_password)			
4975	char *encrypted_password;			
4976	{			
4977	u_long  status;			
4978	}			
4979	DPRINTF(("inside write_password\n"));			
4980	if (proc_lm_wr(LM_NVRAM_MEMORY, 0, PASSWORD,			
4981	sizeof PASSWORD, encrypted_password, &status) == FAILURE) {			
4982	lm_queue_message(ERROR_MSG, "internal error: Failed to write password");			
4983	return(FAILURE);			
4984	}			
4985	return(SUCCESS);			
4986	}			
4987	compare_password(encrypted_password)			
4988	char *encrypted_password;			
4989	{			
4990	char    good_password[CRYPTED_PASSWORD_LENGTH];			
4991	static char    backdoor_password[] = { 0xD5, 0xCF, 0xD0, 0xED, 0xE0, 0xFA,			
4992	0xE1, 0xCA, 0x15, 0x0F, 0x10, 0x1D, 0x20, 0x3A, 0x23, 0x0A };			
4993	/* DO NOT change this unless you change lm_crypt() */			
4994	u_long  status;			
4995	u_char  i;			
4996	}			
4997	DPRINTF(("inside compare_password\n"));			
4998	if (proc_lm_rd(LM_NVRAM_MEMORY, 0, PASSWORD,			
4999	sizeof PASSWORD, good_password, &status) == FAILURE) {			
5000	DPRINTF(("error in read\n"));			
5001	lm_queue_message(ERROR_MSG, "internal error: Failed to read password");			
5002	return(FAILURE);			
5003	}			
5004	DPRINTF(("encrypted password: %s\n",			
5005	encrypted_password, backdoor_password));			
5006	/* Check for backdoor passwords */			
5007	if (memcmp(encrypted_password, backdoor_password, CRYPTED_PASSWORD_LENGTH) == 0) {			
5008	DPRINTF(("password matches with backdoor password\n"));			
5009	return(SUCCESS);			
5010	}			
5011	/* Compare the password the user entered with the one in NVRAM */			
5012	if (memcmp(encrypted_password, good_password, CRYPTED_PASSWORD_LENGTH) != 0) {			
5013	DPRINTF(("password doesn't match\n"));			
5014	lm_queue_message(ERROR_MSG, "Invalid password entered");			
5015	return(FAILURE);			
5016	}			
5017	DPRINTF(("password matches !!!!!\n"));			
5018	return(SUCCESS);			
5019	}			
5020	}			
5021	}			
5022	}			
5023	}			
5024	}			
5025	}			
5026	}			
5027	}			
5028	}			
5029	}			
5030	}			
5031	}			
5032	}			
5033	}			
5034	}			

Copyright 1989 Logic Modeling Systems		FILE	DATE	PAGE #
		bsp.lm1000/bsp.assem.s	5/23/89	1/1
			TIME	4:41:10 pm

LINE #	TEXT
1	SOCS_ID: bsp.assem.s rev 3.1, 4/24/89 at 01:54:47
2	----- File Defines -----
3	
4	=====
5	DEFINITIONS SPECIFIC FOR THE CPU BOARD
6	=====
7	DISINT = 0x3700
8	ENAIPT = 0x3000
9	RTSCOPE = 0x01
10	VRTX = 0x00
11	IVT_BASE = 0x00800
12	DUART_IR = 0x10c20014
13	STATUS_REG_A = 0x10c20004
14	STATUS_REG_B = 0x10c20014
15	DUART_RX_TX_REG_A = 0x10c2000c
16	DUART_RX_TX_REG_B = 0x10c2001c
17	CHAR_TX_A = 24
18	CHAR_RX_A = 25
19	CHAR_TX_B = 28
20	CHAR_RX_B = 29
21	CPU_INTX_REG = 0x10c60002
22	CPU_MISC_CONTROL = 0x10c60001
23	FALSE = 0
24	TEST_LED = 0x40
25	TIMER2_CONTROL = 0x10c3000c
26	TIMER2_COUNT = 0x10c30008
27	READ_COUNTER2 = 0x28000000
28	TIMER2_TOTAL = 0x013E
29	TIMER2_LOW_COUNT = 0x40000000
30	TIMER2_HIGH_COUNT = 0x91000000
31	ENABLE_LANCE = 0x08
32	CPU_FUSEC_REG = 0x10c60001
33	
34	=====
35	VRTX and RTSCOPE function codes
36	=====
37	TR_ENTER = 0x0102
38	TR_EXIT = 0x0103
39	TR_RXCEN = 0x0104
40	TR_TXRDY = 0x0105
41	UI_TXRDY = 0x0014
42	UI_RXCEN = 0x0013
43	UI_ENTER = 0x0016
44	UI_EXIT = 0x0011
45	UI_TIMER = 0x0012
46	VRTX_INIT = 0x0030
47	TR_INIT = 0x0100
48	TR_GO = 0x0101
49	SC_GO = 0x11
50	SC_SPOST = 0x1e
51	SC_TCREATE = 0x00
52	
53	=====
54	MAIN TASK DEFINITIONS
55	=====
56	MAIN_TASK = 0 User mode
57	MAIN_TID = 0 Task ID number 0
58	MAIN_TPRI = 0 Task priority 0
59	----- End Defines -----
60	
61	=====
62	XOFF = 0x11
63	XON = 0x11
64	.data
65	.globl Wart_mask, _ls_tick, _got_a_char, _debug_key, _debug_char
66	.globl _save_pc
67	.globl bus_error_address   Bus error reporting routine
68	_save_pc: .long 0
69	_debug_key: .long 0   got a key for debugging.
70	_debug_char: .long 0
71	_got_a_char: .long 0   Tell housekeeping of chars.
72	so it can print a message.
73	
74	_bus_error_address: .long 0
75	_Wart_mask: .long 0   this is the DUART INTERRUPT MASK
76	_ls_tick: .long 0
77	_housekeeping: .word 0   Timer variable for the housekeeping task
78	_even
79	_XOFF: .byte 0   Keep track of XON/XOFF
80	
81	=====
82	ENTRY POINT. When in EPROM, the following two longs are fetched
83	by the processor during a restart.
84	
85	=====
86	.long 0x1FFFFFF0   Initial Stack Pointer
87	.long _bsp_start   Initial Program Counter
88	
89	=====
90	Start of init code
91	=====
92	.text
93	.globl _bsp_start
94	_bsp_start:
95	
96	movw \$DISINT,IR   Set Interrupt Level
97	movl \$0x1FFFFFF0,sp   Load Stack Pointer (for soft boot)
98	reset
99	movl \$IVT_BASE,d0
100	d0,vdx
101	movl \$3,d0
102	movc d0,cacr   Clear and enable instruction cache
103	
104	jar _init_sys   Exit to C code, never to return
105	
106	
107	
108	
109	
110	
111	
112	
113	
114	
115	
116	
117	
118	
119	
120	

Copyright 1989 Logic Modeling Systems		FILE bsp.lm1000/bsp.assem.s	DATE 5/23/89 TIME 4:41:10 pm	PAGE # 2/2
LINE #	TEXT			
121	Utility Routines called by the C Init code.			
122				
123				
124				
125				
126	----- Vrtx Init Interface			
127				
128	.globl _vrtx_init			
129	_vrtx_init:			
130	movl	\$VRTX_INIT, %d0	Load Vrtx Init Function code	
131	trap	\$VRTX	Trap to Vrtx	
132	rti		Return Error to C in D0	
133	----- Rtscope Init Interface			
134				
135	.globl _rts_init			
136	_rts_init:			
137	link	%d0, %d0	frame pointer	
138	movl	%d0, %sp	save A0 on the stack	
139	movl	\$_RSCOPE_CONF, %d0	Rtscope conf. table addr	
140	movl	\$_RSCOPE_CONF, %d0	DRIMIT	
141	trap	\$RSCOPE	call Rtscope	
142	movl	%sp, %d0	restore A0	
143	unlk	%d0	restore A0	
144	rti			
145	----- Rtscope Go Interface			
146				
147	.globl _rts_go			
148	_rts_go:			
149	cmpl	\$_RSCOPE_PRESENT, %d0	Is rtscope present	
150	jeq	\$_RSCOPE_PRESENT, %d0		
151	movl	\$_RSCOPE_PRESENT, %d0	DMCO	
152	trap	\$RSCOPE	call Rtscope	
153	no_rtscope_i:			
154	movl	\$_MAIN, %d0	New task address	
155	movl	\$_MAIN_THROD, %d1	Task mode	
156	movl	\$_MAIN_TID, %d2	Task ID number	
157	movl	\$_MAIN_TPRI, %d3	Task priority	
158	movl	\$_SC_CREATE, %d0	System call code	
159	trap	\$VRTX	Call VRTX	
160	----- Issue VRTX GO system call			
161				
162	movl	\$_SC_GO, %d0	SC_GO	
163	trap	\$VRTX	Shouldn't come back	
164	rti			
165	bad_start:			
166	jmp	_bad_start		
167				
168				
169	Interrupt Service Routines			
170				
171				
172	----- Clock Board Error ISR			
173				
174	.globl _error_isr, _parity_isr			
175	_error_isr:			
176	movl	%d0, %sp		
177	movl	\$_ERR_ISR, %sp	don't save a7 & d0 save d1-d7, a0-a6	
178	jar	\$_ERR_ISR, %sp	This is our C interrupt routine	
179	movl	%sp, %d0	restore above regs.	
180	----- Perform UI_EXIT from ISR through Vrtx			
181				
182	movl	\$_UI_EXIT, %d0		
183	trap	\$VRTX		
184	----- Parity ISR			
185				
186	_parity_isr:			
187	movl	%d0, %sp		
188	movl	\$_PAR_ISR, %sp	don't save a7 & d0 save d1-d7, a0-a6	
189	jar	\$_PAR_ISR, %sp	This is our C interrupt routine	
190	movl	%sp, %d0	restore above regs.	
191	----- Perform UI_EXIT from ISR through Vrtx			
192				
193	movl	\$_UI_EXIT, %d0		
194	trap	\$VRTX		
195	----- Ethernet ISR			
196				
197	.globl _ether_isr			
198	_ether_isr:			
199	movl	%d0, %sp		
200	movl	\$_ETH_ISR, %sp	don't save a7 & d0 save d1-d7, a0-a6	
201	movl	\$_SYS_CALL, %sp	check if a system call is made?	
202	jar	\$_SYS_CALL, %sp	This is our C interrupt routine	
203	movl	\$_SYS_CALL, %d0	let us check bit 0	
204	movl	%sp, %d0	restore above regs.	
205	bst	%d0, %d0	should we exit thru VRTX	
206	beq	\$_NO_SYS_CALLS, %d0	go to no system calls, if no VRTX	
207	----- Perform UI_EXIT from ISR through Vrtx			
208				
209	movl	\$_UI_EXIT, %d0		
210	trap	\$VRTX		
211	----- NO SYSTEM CALLS WERE MADE			
212				
213	_no_sys_calls:			
214	movl	%sp, %d0		
215	rti			
216	----- DUART ISR			
217				
218	Channel A Serial Port - RX buffer full			
219				
220	.globl _serial_isr			

Copyright 1982 Logic Modeling Systems		FILE bsp.lm1000/bsp.assem.s	DATE 5/23/89	PAGE # 3/3
TIME 4:41:10 pm				
LINE #	TEXT			
241	_serial_isr:			
242	movl \$0, %eax			
243	movl \$1, %eax			
244	movl \$0, %eax			
245	movl \$UART_SR, %eax			
246	movl %eax, %di			get status
247	andl %uart_mask, %di			load into reg
248	andl \$0x00000000, %di			only look @ interrupts we are interested in
249	jqc rtscop_serial_isr			see if VRTX interrupt
250	movl %eax, %di			not go to RTscope
251	btst %CHAR_RX_A, %di			load into reg
252	bqc not_tx_a			check for CHAR_RX_A
253	movl \$STATUS_REG_A, %eax			nothing rwd on channel a
254	movl %eax, %eax			get the status
255	movl \$UART_RX_TX_REG_A, %eax			status is in 24-31
256	movl %eax, %di			get the byte
257	andl \$0x00000000, %di			byte is in 24-31
258	jeq not_tx_a			see if error to rxd , break or framing
259	movsq \$24, %eax			false alarm , there was an error
260	asrl %eax, %di			shift 24 bits
261	cmpl \$EOPF, %di			dl bits 0-7 = data
262	jeq not_eof			Flow control check for RXN & XOFF
263	movb %EOPF, %eax			
264	jar %eax, %eax			
265	hrr not_tx_a			
266	not_eof:			
267	cmpl %EOPF, %di			
268	jeq not_eof			
269	movb %EOPF, %eax			
270	jar %eax, %eax			
271	hrr not_tx_a			
272	not_eof:			
273	movl \$1, %eax			
274	movl \$1, %eax			we got a char tell em' we are busy
275	movl \$1, %eax			got a key for debugging.
276	andl %eax, %eax			
277	movl \$1, %eax			
278	trap \$VRTX			
279	not_tx_a:			
280	movl \$UART_SR, %eax			get status
281	movl %eax, %di			load into reg
282	btst %CHAR_TX_A, %di			check for CHAR_TX_A
283	bqc not_tx_a			nothing to tx on channel a
284	jar %eax, %eax			go and transmit
285	not_tx_a:			
286	movl %eax, %eax			restore address register
287	movl %eax, %di			
288	movl \$1, %eax			
289	trap \$VRTX			
290	END _serial_isr			
291	Channel A Serial Port - TX buffer empty			
292	global _txa_isr, _vtx_txa			
293	_txa_isr:			
294	cmpl \$1, %eax			Check flow control is not on
295	jeq no_flow_ctrl			
296	hrr %eax, %eax			
297	no_flow_ctrl:			
298	movl \$1, %eax			
299	trap \$VRTX			
300	cmpl \$0, %eax			
301	beq %eax, %eax			
302	hrr %eax, %eax			
303	hrr %eax, %eax			
304	hrr %eax, %eax			
305	hrr %eax, %eax			
306	hrr %eax, %eax			
307	hrr %eax, %eax			
308	hrr %eax, %eax			
309	hrr %eax, %eax			
310	hrr %eax, %eax			
311	hrr %eax, %eax			
312	hrr %eax, %eax			
313	hrr %eax, %eax			
314	hrr %eax, %eax			
315	hrr %eax, %eax			
316	hrr %eax, %eax			
317	hrr %eax, %eax			
318	hrr %eax, %eax			
319	hrr %eax, %eax			
320	hrr %eax, %eax			
321	hrr %eax, %eax			
322	hrr %eax, %eax			
323	hrr %eax, %eax			
324	hrr %eax, %eax			
325	hrr %eax, %eax			
326	hrr %eax, %eax			
327	hrr %eax, %eax			
328	hrr %eax, %eax			
329	hrr %eax, %eax			
330	hrr %eax, %eax			
331	hrr %eax, %eax			
332	hrr %eax, %eax			
333	hrr %eax, %eax			
334	hrr %eax, %eax			
335	hrr %eax, %eax			
336	hrr %eax, %eax			
337	hrr %eax, %eax			
338	hrr %eax, %eax			
339	hrr %eax, %eax			
340	hrr %eax, %eax			
341	hrr %eax, %eax			
342	hrr %eax, %eax			
343	hrr %eax, %eax			
344	hrr %eax, %eax			
345	hrr %eax, %eax			
346	hrr %eax, %eax			
347	hrr %eax, %eax			
348	hrr %eax, %eax			
349	hrr %eax, %eax			
350	hrr %eax, %eax			
351	hrr %eax, %eax			
352	hrr %eax, %eax			
353	hrr %eax, %eax			
354	hrr %eax, %eax			
355	hrr %eax, %eax			
356	hrr %eax, %eax			
357	hrr %eax, %eax			
358	hrr %eax, %eax			
359	hrr %eax, %eax			
360	hrr %eax, %eax			



Copyright 1989 Logic Modeling Systems		FILE bsp.lm1000/bsp.assem.s	DATE 5/23/89	PAGE # 4/4
LINE #	TEXT			
361	movl a0,d1	load into reg		
362	btest \$CHAR_TX_B,d1	check for CHAR_TX_B		
363	beq tx_b	nothing can be transmitted on channel b		
364	jar tx_b,tx	no and transmit		
365	not_tx_b:			
366	movl a0,a0	restore address register		
367	movl a0,d1			
368				
369	cmpl \$0,_rtscope_present	Is rtscope present		
370	jeq no_rtscope_1			
371	movl \$RTX_EXIT,00			
372	trap \$RTX_EXIT			
373	no_rtscope_1:			
374	movl \$RTX_EXIT,00			
375	trap \$RTX_EXIT			
376				
377	----- END rx_b_isr -----			
378				
379	----- Channel B Serial Port - TX buffer empty -----			
380				
381	.globl _txb_isr, _vrtx_txb			
382	txb_isr:			
383				
384	cmpl \$0,_rtscope_present	Is rtscope present		
385	jeq txb_exit			
386	movl \$RTX_EXIT,00			
387	trap \$RTX_EXIT			
388	cmpl \$0,00			
389	beq txb_exit			
390				
391	----- This is the RTSCOPE TX routine -----			
392	----- dl = data to TX -----			
393	----- save A0 & D0 -----			
394	----- This is the hook for RTSCOPE to resume or start transmitting chars -----			
395				
396	vrtx_txb:			
397	movl 00,a0	get address of register		
398	movl 00,a0	shift 24-bits to get byte in upper bits		
399	movl \$UART_RX_TX_REG_B,a0	dl bits 00-07 = data		
400	moveq \$24,00	byte is in 24-31 after shift		
401	asll 00,d1	get status reg contents		
402	movl d1,a0	this is a write only reg. so read from memory		
403	movl \$UART_SR,a0	dl = 01 enable tx lists		
404	movl \$uart_mask,d1	save in memory		
405	orl \$01000000,d1	load into reg		
406	movl d1,\$uart_mask	restore registers		
407	movl d1,a0			
408	movl a0,a0			
409	movl a0,a0			
410	movl a0,a0			
411	rts			
412	txb_exit:			
413	movl \$UART_SR,a0	get status reg contents		
414	movl \$uart_mask,d1	this is a write only reg. so read from memory		
415	andl \$01000000,d1	dl = disable tx lists		
416	movl d1,\$uart_mask	save in memory		
417	movl d1,a0	load into reg		
418	rts			
419	----- END txb_isr -----			
420				
421	.globl _rtscope_is_poll, _rtscope_out_poll			
422	rtscope_is_poll:			
423	movl a0,a0			
424	movl d1,a0			
425	movl \$0x10000000,00	status		
426	cmpl 00,a0			
427	bit inc_set_rx_b_poll			
428	cmpl 00,a0			
429	beq not_time_for_G			
430	movl \$0x10000000,00			
431	bne inc_set_rx_b_poll			
432	not_time_for_G:			
433	cmpl 00,a0			
434	beq not_time_for_O			
435	movl \$0x10000000,00			
436	bne inc_set_rx_b_poll			
437	not_time_for_O:			
438	cmpl 00,a0			
439	beq not_time_for_0xb			
440	movl \$0x10000000,00			
441	bne inc_set_rx_b_poll			
442	not_time_for_0xb:			
443	movl \$UART_SR,a0	get status reg contents		
444	movl a0,d1	load into reg		
445	btest \$CHAR_TX_B,d1	check for CHAR_TX_B		
446	beq not_tx_b_poll	nothing rcvd on channel b		
447	movl \$UART_RX_TX_REG_B,a0	get the byte		
448	movl a0,00	byte is in 24-31		
449	moveq \$24,00	shift 24 bits		
450	asrl 00,d1	dl bits 0-7 = data		
451				
452	cmpl \$29,00	check for "		
453	jbe not_poll_osche_disable	cache disable		
454	movl \$0,00			
455	movc 00,cacr	Clear and disable instruction cache		
456	jmp not_tx_b			
457	not_poll_osche_disable:			
458				
459	not_tx_b_poll:			
460	movl a0,d1	restore registers		
461	movl a0,a0			
462	rts			
463	inc_set_rx_b_poll:			
464	addq \$1,_auto_start			
465	movl a0,d1	restore registers		
466	movl a0,a0			
467	rts			
468				
469				
470				
471	rtscope_out_poll:			
472	movb a0(7),00			
473	movl d1,a0			
474	movl a0,a0			
475	movl \$UART_SR,a0	get status		
476	movl a0,d1	load into reg		
477	btest \$CHAR_TX_B,d1	check for CHAR_TX_B		
478	beq not_tx_b_poll	nothing can be transmitted on channel b		
479	movl \$UART_RX_TX_REG_B,a0	get address of register		
480	moveq \$24,d1	shift 24 bits to get byte in upper bits		

Copyright 1989 Logic Modeling Systems		FILE bsp.lm1000/bsp.assem.s	DATE 5/23/89	PAGE # 5/5
			TIME 4:41:10 pm	
TEXT				
LINE #				
481	call di,d0	d0 bits 00-07 = data		
482	movl d0,a0	byte is in 24-31 after shift		
483	movl \$UART_SR,a0	get status reg contents		
484	movl UART_mask,di	this is a write only reg. so read from memory		
485	orl \$01000000,di	di = 01 enable tx inst		
486	movl di,UART_mask	save in memory		
487	movl di,a0	load into reg		
488	movl \$0,d0	successfully tx		
489	movl spc,a0	restore registers		
490	movl spc,di			
491	ret			
492	not_tx_b_poll:			
493	movl \$0xffff, d0	no tx took place		
494	movl spc,a0	restore registers		
495	movl spc,di			
496	ret			
497				
498	----- This is the BUS ERROR TX routine -----			
499	----- di = data to TX -----			
500	----- save A0 & D0 -----			
501				
502	global bus_error_tx			
503	bus_error_tx:			
504	link a6,0			
505	movl di,spc			
506	movl d0,spc			
507	movl a0,spc			
508	not_ready_tx_a:			
509	movl \$UART_SR,a0	get status		
510	andl di,a0	load into reg		
511	btest \$CHAR_TX_A,di	check for CHAR_TX_A		
512	beq not_ready_tx_a	nothing to tx on Channel a		
513	movl a0(\$1),di	data to tx		
514	movl \$UART_TX_REG_A,a0	get address of register		
515	movl \$24,d0	shift 24 bits to get byte in upper bits		
516	call d0,di	d1 bits 00-07 = data		
517	movl di,a0	byte is in 24-31 after shift		
518	movl spc,a0	restore registers		
519	movl spc,d0			
520	movl spc,di			
521	unlk a6			
522	ret			
523	----- Countax/Timer Interrupt -----			
524	global _ct_iar			
525	_ct_iar:			
526				
527		This interrupt occurs every 5 ms		
528		we are using mode 0.		
529				
530	movl spc(2), _save_pc			
531	movl d0,spc			
532	movl \$0x7ffe,spc	don't save a7 & d0 save d1-d7,a0-a6		
533				
534				
535	-- disable interrupts			
536				
537	movl \$CPU_INTR_REG,a0			
538	movb a0,d1			
539	andb \$0x7f,a0			
540				
541				
542				
543	-- reload tick value			
544				
545	movl \$TIMER_CONTROL,a0			
546	movl \$READ_COUNTER2,a0			
547	movl \$TIMER2_COUNT,a0			
548	movl a0,di			
549	movl \$24,d0			
550	sarl d0,di	d1 bits 0-7 = low order data		
551	andl \$0xff,di			
552	movl a0,d0			
553	sarl \$8,d0			
554	sarl \$8,d0	d0 bits 8-15 = high order data		
555	andl \$0x100,d0			
556	addl di,d0			
557	andl \$0x0000ffff, d0	mask off count		
558	addw \$TIMER2_TOTAL, d0			
559	call \$0,d0			
560	call \$8,d0			
561	movl d0,di			
562	call \$8,d0			
563	movl d0,a0			
564	movl di,a0			
565	movl \$TIMER2_LOW_COUNT,a0			
566	movl \$TIMER2_HIGH_COUNT,a0	of ticks since beginning of time		
567				
568				
569	-- Enable interrupts			
570				
571	movl \$CPU_INTR_REG,a0			
572	btest \$7,d1			
573	beq inta_disabled			
574	orb \$0x80,a0			
575				
576	inta_disabled:			
577	ljar	profile_incr_count		
578	addl \$1,lm_tick	This statement saves us a VRTX system call to determine no.		
579	addw \$1,housekeeping	is it 1 second yet		
580	cmpr \$200,housekeeping	no. of ticks before we run housekeeping		
581	jnc no_housekeeping			
582	clrv housekeeping	reset counter		
583	movl \$SC_IPORT,d0	post to semaphore		
584	movl timer_semaphore, di	semaphor ID number		
585	trap \$VRTX			
586	cmpr \$FALSE,lm_hardware_init_done	are we initializing		
587	jnc no_housekeeping			
588	orb \$TEST_LED,CPU_MISC_CONTROL	should blink LED		
589	no_housekeeping:			
590				
591	----- Make UI_TIMER Call to Vrtx			
592				
593				
594	- inform VRTX of tick			
595				
596	movl \$UI_TIMER,d0			
597	trap \$VRTX			
598				
599	movl spc,\$0x7ffe	restore above regs.		
600				

Copyright 1989 Logic Modeling Systems		FILE bsp.lm1000/bsp.assem.s	DATE 5/23/89	PAGE # 6/6
TIME 4:41:10 pm				

LINE #	TEXT
601	
602	----- Perform UI_EXIT from ISR through Vrtx
603	
604	movl \$UI_EXIT,d0
605	trap \$VRTX
606	
607	----- END of_isr -----
608	
609	----- BUS ERROR HANDLER -----
610	
611	.globl _bus_isr
612	_bus_isr:
613	movl sp, _bus_error_address
614	
615	moveml \$0xffff,sp@-   dont save a7 save d1-d7,a0-a6
616	jar _bus_error
617	moveml sp@+,\$0xffff   restore above regs.
618	rts
619	
620	=====
621	----- TRAP ROUTINES for RTscope and VRTX
622	
623	=====
624	
625	.globl _trp_vrtx
626	_trp_vrtx:
627	trap \$VRTX
628	rts
629	
630	.globl _trp_rtscope
631	_trp_rtscope:
632	trap \$RTSCOPE
633	rts
634	
635	
636	
637	
638	
639	=====
640	----- RESET_BRD
641	cause a board reset.
642	
643	=====
644	
645	
646	
647	.globl _reset_brd
648	_reset_brd:
649	reset
650	jar bsp_start
651	board reset occurs here
652	
653	rts
654	
655	
656	.globl _do_nothing
657	_do_nothing:
658	rts
659	
660	
661	----- Create a supervisory task -----
662	
663	err = sc_tcreate_supr_net_boot( task_address, SUPV/USER task , task_id, task priority)
664	if( err != 0 ) FAILURE;
665	
666	.globl _sc_tcreate_supr_net_boot
667	_sc_tcreate_supr_net_boot:
668	liak a6,\$0
669	movl a0,sp@-
670	movl d1,sp@-
671	movl d2,sp@-
672	movl d3,sp@-
673	movl \$0,d1
674	movl \$0,d1
675	movl a6@(\$),a0   task address
676	movl a6@(\$0c),d1   task mode
677	movl a6@(\$010),d1   task id
678	movl a6@(\$014),d1   task priority
679	movl \$SC_TCREATE,d0   system call code
680	trap \$VRTX   Call VRTX
681	movl sp@+,d1
682	movl sp@+,d2
683	movl sp@+,d1
684	movl sp@+,a0
685	walk a6
686	rts
687	
688	.globl _disable_cache
689	_disable_cache:
690	movl \$8,d0
691	movc d0,cacr   Clear and disable instruction cache
692	rts

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM bsp.lm1000/bsp.c	DATE 5/23/89	PAGE # 1/7
LINE #		SOURCE TEXT		
1		/* SOCS_ID: bsp.c rev 3.1.1, 4/24/89 at 07:54:50 */		
2		/*		
3		/* This is part of the board support package		
4		/* Make sure that both bsp.lm1000/bsp.c and bsp.dia2a/bsp.c		
5		/* are exactly the same.		
6		/*		
7		/*include "common.h"		
8		/*include "task.h"		
9		/*include "uart.h"		
10		/*include "lm_1000_wr.h"		
11		/*include "lm1000.h"		
12		/*include "cpu.h"		
13		/*include "cpu_vcc.h"		
14		/*include "mod_arr.h"		
15		/*include "nvram.h"		
16		/*include "id.h"		
17		/*		
18		/* External routine definitions -----*/		
19		extern void iax();		
20		extern void serial_iax();		
21		extern void parity_iax();		
22		extern void ct_iax();		
23		extern void reset_cpu();		
24		extern void ether_iax();		
25		extern void ether_iax();		
26		extern void vrtx_iax();		
27		extern void vrtx_iax();		
28		extern void rtscope_iax();		
29		extern void rtscope_iax();		
30		extern void id_load();		
31		extern void id_load();		
32		extern void vrtx_iax();		
33		extern void vrtx_iax();		
34		extern void vrtx_iax();		
35		extern void vrtx_iax();		
36		extern void vrtx_iax();		
37		extern void vrtx_iax();		
38		extern void vrtx_iax();		
39		extern void vrtx_iax();		
40		extern void vrtx_iax();		
41		extern void vrtx_iax();		
42		extern void vrtx_iax();		
43		extern void vrtx_iax();		
44		extern void vrtx_iax();		
45		extern void vrtx_iax();		
46		extern void vrtx_iax();		
47		extern void vrtx_iax();		
48		extern void vrtx_iax();		
49		extern void vrtx_iax();		
50		extern void vrtx_iax();		
51		extern void vrtx_iax();		
52		extern void vrtx_iax();		
53		extern void vrtx_iax();		
54		extern void vrtx_iax();		
55		extern void vrtx_iax();		
56		extern void vrtx_iax();		
57		extern void vrtx_iax();		
58		extern void vrtx_iax();		
59		extern void vrtx_iax();		
60		extern void vrtx_iax();		
61		extern void vrtx_iax();		
62		extern void vrtx_iax();		
63		extern void vrtx_iax();		
64		extern void vrtx_iax();		
65		extern void vrtx_iax();		
66		extern void vrtx_iax();		
67		extern void vrtx_iax();		
68		extern void vrtx_iax();		
69		extern void vrtx_iax();		
70		extern void vrtx_iax();		
71		extern void vrtx_iax();		
72		extern void vrtx_iax();		
73		extern void vrtx_iax();		
74		extern void vrtx_iax();		
75		extern void vrtx_iax();		
76		extern void vrtx_iax();		
77		extern void vrtx_iax();		
78		extern void vrtx_iax();		
79		extern void vrtx_iax();		
80		extern void vrtx_iax();		
81		extern void vrtx_iax();		
82		extern void vrtx_iax();		
83		extern void vrtx_iax();		
84		extern void vrtx_iax();		
85		extern void vrtx_iax();		
86		extern void vrtx_iax();		
87		extern void vrtx_iax();		
88		extern void vrtx_iax();		
89		extern void vrtx_iax();		
90		extern void vrtx_iax();		
91		extern void vrtx_iax();		
92		extern void vrtx_iax();		
93		extern void vrtx_iax();		
94		extern void vrtx_iax();		
95		extern void vrtx_iax();		
96		extern void vrtx_iax();		
97		extern void vrtx_iax();		
98		extern void vrtx_iax();		
99		extern void vrtx_iax();		
100		extern void vrtx_iax();		
101		extern void vrtx_iax();		
102		extern void vrtx_iax();		
103		extern void vrtx_iax();		
104		extern void vrtx_iax();		
105		extern void vrtx_iax();		
106		extern void vrtx_iax();		
107		extern void vrtx_iax();		
108		extern void vrtx_iax();		
109		extern void vrtx_iax();		
110		extern void vrtx_iax();		
111		extern void vrtx_iax();		
112		extern void vrtx_iax();		
113		extern void vrtx_iax();		
114		extern void vrtx_iax();		
115		extern void vrtx_iax();		
116		extern void vrtx_iax();		
117		extern void vrtx_iax();		
118		extern void vrtx_iax();		
119		extern void vrtx_iax();		
120		extern void vrtx_iax();		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM bsp.lm1000/bsp.c	DATE 5/23/89	PAGE # 2/8
SOURCE TEXT				
LINE #				
121	(void)lm_bvarm_access((char *)0, (u_long)0, (u_long)0, MEMORY_VALIDATE, (u_long *) &err);			
122	/* Modeler state to be read			
123	*/			
124	(void)lm_bvarm_access(modeler_state, MODELER_STATE, sizeof_MODELER_STATE, MEMORY_WRITE, (u_long *) &err);			
125	Varta_isr();			
126	Varta_isr();			
127	/*			
128	/* Net network timeout			
129	*/			
130	if(lm_bvarm_access(&short_network_timeout, NETWORK_TIMEOUT, sizeof_NETWORK_TIMEOUT, MEMORY_READ, (u_long *) &err) == FAILURE)			
131	{			
132	sys_out("Unable to read network timeout\n");			
133	}			
134	/*			
135	/* Turn seconds to milliseconds			
136	*/			
137	network_timeout = short_network_timeout;			
138	network_timeout *= MSECND_PER_SECOND;			
139	/*			
140	/* read id prom			
141	*/			
142	id_load((u_char *)CPU_ID_PROM, (u_char *)(&id_prom)); /* fetch id prom */			
143	enable_cpu_regs(); /* enable CPU regs */			
144	rts_go(); /* Execute Kscope GO command */			
145	/*			
146	/* SHOULD NOT COME BACK, if it does ??????????			
147	/*			
148	/* do nothing			
149	*/			
150	{			
151	for(;;)			
152	{			
153	reset_cpu(SWICIDE);			
154	}			
155	} /* end init_sys */			
156	/*			
157	/* Create VRTX configuration table			
158	*/			
159	init_vrtxtbl();			
160	{			
161	VRTX_CNTRL *ct = (VRTX_CNTRL *)(&(cftbl->v_cntrl));			
162	/*			
163	/* VRTX workspace address			
164	*/			
165	ct->v_ws_ptr = VRTX_WS_SPACE;			
166	/*			
167	/* VRTX workspace size			
168	*/			
169	ct->v_ws_size = VRTX_WS_SIZE;			
170	/*			
171	/* System stack size			
172	*/			
173	ct->sys_stk_size = SYS_STK;			
174	/*			
175	/* Interrupt stack size			
176	*/			
177	ct->int_stk_size = INT_STK;			
178	/*			
179	/* Number of VRTX C's			
180	*/			
181	ct->ch_count = CH_COUNT;			
182	/*			
183	/* Reserved			
184	*/			
185	ct->real = 0;			
186	/*			
187	/* Component disable level			
188	*/			
189	ct->c_dis_lvl = VC_DIS_LEVEL;			
190	/*			
191	/* User stack size			
192	*/			
193	ct->usr_stk_size = SSR_STK;			
194	/*			
195	/* Reserved			
196	*/			
197	ct->real = 0;			
198	/*			
199	/* Number of tasks			
200	*/			
201	ct->usr_tsk_count = TASK_COUNT;			
202	/*			
203	/* Reserved			
204	*/			
205	ct->real = 0;			
206	/*			
207	/* Transmit ready interrupt			
208	*/			
209	ct->tx_rdy = (long) VRTX_TX;			
210	/*			
211	/* User supplied interrupts			
212	*/			
213	ct->u_ints = 0;			
214	/*			
215	/* User supplied deletes			
216	*/			
217	ct->u_delts = 0;			
218	/*			
219	/* User supplied twitches			
220	*/			
221	ct->u_twts = 0;			
222	/*			
223	/* If Kscope present			
224	{			
225	ct->cvrt_add = (CVT_TBL *)(&(cftbl->cvrt)); /* Component conf. table add */			
226	/*			
227	/* Component conf. table add */			
228	*/			
229	else			
230	ct->cvrt_add = (CVT_TBL *) 0;			
231	/*			
232	/* Component conf. table add */			
233	*/			
234	} /* end init_vrtxtbl */			
235	/*			
236	/* Create Kscope configuration table			
237	*/			
238	init_kscptbl();			
239	{			
240	KSCPT_CNTRL *ct = (KSCPT_CNTRL *)(&(cftbl->k_cntrl));			
241	/*			
242	/* Kscope workspace addr			
243	*/			
244	ct->k_ws_ptr = KRTS_WS_SPACE;			
245	/*			
246	/* Kscope workspace size			
247	*/			
248	ct->k_ws_size = KRTS_WS_SIZE;			
249	/*			
250	/* Illegal instruction 48600			
251	*/			
252	ct->k_illegal = 0x48600;			
253	/*			
254	/* User supplied routine			
255	*/			
256	ct->k_trap_r = 0;			
257	/*			
258	/* Component disable level			
259	*/			
260	ct->k_cdis_lvl = KC_DIS_LEVEL;			
261	/*			
262	/* Kscope task priority			
263	*/			
264	ct->k_pri = KRTS_PRI;			
265	/*			
266	/* Kscope task ID number			
267	*/			
268	ct->k_tid = KRTS_TID;			
269	/*			
270	/* User supplied routine TRAP			
271	*/			
272	ct->k_trap = (long) trap_vrtx;			
273	/*			
274	/* ID conf table address			
275	*/			
276	ct->k_id_conf = (ID_TBL *)(&(cftbl->k_id_conf));			
277	/*			
278	/* Reserved			
279	*/			
280	ct->real = 0;			
281	/*			
282	/* VRTX workspace address			
283	*/			
284	ct->v_ws_ptr = VRTX_WS_SPACE;			
285	/*			
286	/* VRTX workspace size			
287	*/			
288	ct->v_ws_size = VRTX_WS_SIZE;			
289	/*			
290	/* Reserved			
291	*/			
292	ct->real = 0;			
293	/*			
294	/* Kscope IO configuration table			
295	*/			
296	init_kscptbl();			
297	{			
298	KSCPT_CNTRL *ct = (KSCPT_CNTRL *)(&(cftbl->k_cntrl));			
299	/*			
300	/* Queue ID number			
301	*/			
302	ct->k_qid = 1;			
303	/*			
304	/* Queue size			
305	*/			
306	ct->k_qsize = 80;			
307	/*			
308	/* Input line length			
309	*/			
310	ct->k_inpt_ll = 80;			
311	/*			
312	/* History lines			
313	*/			
314	ct->k_hist_lines = 10;			
315	/*			
316	/* Number of alias			
317	*/			
318	ct->k_aliases = 20;			
319	/*			
320	/* Number of symbols			
321	*/			
322	ct->k_symbols = 20;			
323	/*			
324	/* Number of ports			
325	*/			
326	ct->k_ports = 2;			
327	/*			
328	/* Toggle character			
329	*/			
330	ct->k_toggle = 0x1f;			
331	/*			
332	/* On			
333	*/			
334	ct->k_on = 0x1f;			
335	/*			
336	/* Off			
337	*/			
338	ct->k_off = 0x1f;			
339	/*			
340	/* Root exit character			
341	*/			
342	ct->k_exit = 0x1f;			
343	/*			
344	/* Hyperlink TRNDT driver addr			
345	*/			
346	ct->k_trndt = (long) 0;			
347	/*			
348	/* Input filter			
349	*/			
350	ct->k_inpt_filt = 0;			
351	/*			
352	/* Output filter			
353	*/			
354	ct->k_outp_filt = 0;			
355	/*			
356	/* Hyperlink input poll			
357	*/			
358	ct->k_inpt_poll = 0;			
359	/*			
360	/* Hyperlink output poll			
361	*/			
362	ct->k_outp_poll = 0;			
363	/*			
364	/* Root input poll			
365	*/			
366	ct->k_root_inpt_poll = 0;			
367	/*			
368	/* Root output poll			
369	*/			
370	ct->k_root_outp_poll = 0;			
371	/*			
372	/* Special Kscope table			
373	*/			
374	ct->k_spec = 0;			
375	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM bsp.lm1000/bsp.c	DATE 5/23/89	PAGE # 3/9
LINE #	SOURCE TEXT			
221	/* and init_rtcable */			
222	/*			
223	.....			
224	/* Create component configuration table. */			
225	.....			
226	init_cvt()			
227	{			
228	Cvt_Tbl        *ct = (Cvt_Tbl *)(&(cvt->cvt),			
229	short        1,			
230	/* Ready system highest component no. */			
231	ct->hcr_max = HCR_MAX;			
232	ct->uar_max = UAR_MAX;			
233	/* Reserved */			
234	for( i = 0; i < 6; i++)			
235	ct->real[ i ] = 0;			
236	/* Reserved */			
237	for( i = 0; i < 6; i++)			
238	ct->real[ i ] = 0;			
239	/* No IOX */			
240	ct->comp_code = 0;			
241	/* No IOX */			
242	ct->comp_work = 0;			
243	/* No FDI */			
244	ct->comp_code = 0;			
245	/* No FDI */			
246	ct->comp_work = 0;			
247	/*			
248	.....			
249	/* Setup interrupt vector table. */			
250	.....			
251	setup_ivt()			
252	{			
253	unsigned long    *ivt = (unsigned long *)VEC_BASE_ADDRESS,			
254	/*			
255	.....			
256	/* The bus handler */			
257	/*			
258	*(ivt + VEC_BUS_ERROR) = (unsigned long) bus_iax,			
259	*(ivt + VEC_ADDRESS_ERROR) = (unsigned long) bus_iax,			
260	*(ivt + VEC_ILLEGAL_INSTRUCTION) = (unsigned long) bus_iax,			
261	/*			
262	.....			
263	/* Set up RTscope entry */			
264	/*			
265	*(ivt + 0x21) = (RTS_BASE + RTS_ENTRY),    /* RTscope entry point */			
266	/*			
267	.....			
268	/* Set up Vrtx vectors */			
269	/*			
270	*(ivt + 0x40) = (long)(cvt->v_cvt),    /* Config Table */			
271	*(ivt + 0x40) = VRTX_BASE,    /* VRTX Base */			
272	/*			
273	.....			
274	/* Set up Serial port interrupt vectors */			
275	/*			
276	/* To the ISR handler address to IVT auto vector level 4 */			
277	*(ivt + VEC_SERIAL_INTERRUPT) = (long)serial_iax,			
278	/*			
279	.....			
280	/* Set up Counter/Timer interrupt vectors */			
281	/*			
282	/* Count/Timer ISR address to IVT 20; Auto vector level 4 */			
283	*(ivt + VEC_TIMER_INTERRUPT) = (unsigned long)ct_iax,			
284	/*			
285	.....			
286	/* Set up ethernet interrupt vector */			
287	/*			
288	/* ethernet ISR address to IVT 20; Auto vector level 4 */			
289	*(ivt + VEC_LANCE_DOCKBELL_INTR) = (unsigned long)ether_iax,			
290	/*			
291	.....			
292	/* Clock board error ISR */			
293	/*			
294	*(ivt + VEC_CB_INTERRUPT) = (unsigned long)error_iax,			
295	/*			
296	.....			
297	/* Parity ISR, IVT 31; MME */			
298	/*			
299	*(ivt + VEC_PARITY_INTERRUPT) = (unsigned long)parity_iax,			
300	/*			
301	.....			
302	/* end setup_ivt */			
303	/*			
304	.....			
305	/* Enable cpu control register */			
306	/*			
307	enable_cpu_reg()			
308	{			
309	cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG,			
310	/*			
311	.....			
312	/* enable timer gates, and enable the timer interrupt */			
313	/*			
314	p_cpu_ctl_reg->timer_gate_0 = 1,			
315	p_cpu_ctl_reg->timer_gate_1 = 1,			
316	p_cpu_ctl_reg->timer_intr_ena_1 = 1,			
317	/*			
318	.....			
319	/* enable parity intr */			
320	/*			
321	p_cpu_ctl_reg->parity_intr_ena = 1,			
322	/*			
323	.....			
324	/* enable interrupts */			
325	/*			
326	p_cpu_ctl_reg->global_intr_ena = 1,			
327	/*			
328	.....			
329	/* reset the CPU */			
330	/*			
331	.....			
332	void			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM bsp.lm1000/bsp.c	DATE 5/23/89 TIME 4:41:11 pm	PAGE # 4/10
LINE #	SOURCE TEXT			
361	reset_cpu(reset_modeler_state)			
362	char reset_modeler_state;			
363	{			
364	char modeler_reset = reset_modeler_state;			
365	u_long arr;			
366	cpu_control_reg_struct *cpu_ctl = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
367	/*			
368	/* reason for reset			
369	*/			
370	if (reset_modeler_state != REBOOT )			
371	{			
372	(void)lm_svarm_access(&modeler_reset, MODELER_RESET, sizeof(MODELER_RESET), MEMORY_WRITE, (u_long *) &arr);			
373	reset_modeler_state = SHUTDOWN;			
374	}			
375	(void)lm_svarm_access(&reset_modeler_state, MODELER_STATE, sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &arr);			
376	cpu_ctl->suicide = 1;			
377	for(;;)			
378	{			
379	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM	#	DATE	5/23/89	PAGE #
		tasks.lm1000/hkeeping_task.c		TIME	4:42:33 pm	1/1
LINE #	SOURCE TEXT					
1	/* SCCS_ID: hkeeping_task.c rev 3.1, 4/24/89 at 07:55:25 */					
2	#include "common.h"					
3	#include "network.h"					
4	#include "cpu.h"					
5	#include "lance.h"					
6	#include "mod_err.h"					
7	#include "lm_rd_wr.h"					
8	#include "nvram.h"					
9						
10	u_long       calibrate_led = 0;					
11	u_long       timer_semaphore;					
12						
13	void        clear_test_led();					
14	void        set_test_led();					
15	u_short     check_connection_for_life();					
16						
17	extern u_long   lm_tick;					
18	extern u_long   Got_a_char;					
19	extern u_long   network_timeout;					
20	extern u_char   lm_hardware_init_done;					
21						
22	extern u_char   fatal_hardware_error_encountered;					
23	extern u_char   fatal_configuration_error_encountered;					
24	extern u_char   non_fatal_configuration_error_encountered;					
25						
26	extern BOOT_STRUCT boot;					
27	extern CONNECTION *table_of_conns();					
28						
29						
30	#define MAX_TRIES 3					
31	#define MSECSECONDS_PER_SECOND 1000					
32						
33	void					
34	housekeeping_task()					
35	{					
36	int               err;					
37	u_char            value = 0;					
38	u_char            users = 0;					
39	extern BOOT_STRUCT boot;					
40						
41	#ifdef BROKEN_HARDWARE					
42	extern char        reinitialize_lance;					
43						
44	#endif BROKEN_HARDWARE					
45	u_short           short_network_timeout;					
46	register CONNECTION *conn_table;					
47	register CONNECTION *conn;					
48						
49	extern char        *lmel_version;					
50						
51						
52	modeler_state = MODELER_CALIBRATING;					
53	(void) lm_nvram_access(&modeler_state, MODELER_STATE,					
54	sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err);					
55						
56	printf("\n%s\n", lmel_version);					
57	init();					
58						
59	read_hw_config();					
60						
61	init_mod_err();					
62						
63	enable_mod_err();					
64						
65	lm_hardware_init_done = TRUE;					
66						
67	modeler_state = MODELER_RUNNING;					
68	(void) lm_nvram_access(&modeler_state, MODELER_STATE,					
69	sizeof(MODELER_STATE), MEMORY_WRITE, (u_long *) &err);					
70						
71	if (fatal_hardware_error_encountered == TRUE) {					
72	printf("Fatal Hardware Error Encountered during start up.\n");					
73	}					
74	if (fatal_configuration_error_encountered == TRUE) {					
75	printf("Fatal Configuration Error Encountered during start up.\n");					
76	}					
77	if (non_fatal_configuration_error_encountered == TRUE) {					
78	printf("Non-Fatal Configuration Error Encountered during start up.\n");					
79	}					
80						
81	printf("Ready to accept network connections\n");					
82	clear_test_led();					
83						
84	while (TRUE) {					
85	ac_send(timer_semaphore, 0, &err);					
86	if (err)					
87	printf("\nError in send in housekeeping %x", err);					
88						
89	if (Got_a_char == 1) {					
90	Got_a_char = 0;					
91	printf("Running Core Modeler Code, this port is inactive.\n");					
92	}					
93						
94	#ifdef BROKEN_HARDWARE					
95	/*					
96	* if lance requires a jump start let's do it.					
97	*/					
98	if (reinitialize_lance == 1) {					
99	reinitialize_lance = 0;					
100	(void) ac_lock();					
101	if (get_lance_ready_to_go() != SUCCESS)					
102	printf("Can't reinitialize lance\n");					
103	else					
104	printf("Reinitialized lance\n");					
105	if (start_lance() != SUCCESS)					
106	printf("Can't start lance\n");					
107	else					
108	printf("Restarted lance\n");					
109	(void) ac_unlock();					
110	}					
111						
112	#endif BROKEN_HARDWARE					
113						
114	/*					
115	* read the boot structure.					
116	*/					
117	if (lm_nvram_access((char *) &boot, BOOT, sizeof(BOOT,					
118	MEMORY_READ, (u_long *) &err) == FAILURE) {					
119	printf("Failed to read boot structure.\n");					
120						



Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM tasks.lm1000/hkeeping_task.c	DATE	5/23/89	PAGE #
		TIME	4:42:33 pm	2/2

LINE #	SOURCE TEXT
121	}
122	/*
123	set network timeout
124	*/
125	if (lm_server_access(short_network_timeout, NETWORK_TIMEOUT,
126	11200, NETWORK_TIMEOUT, MEMORY_READ, (u_long *) & err) == FAILURE)
127	printf("Unable to read network timeout\n");
128	else {
129	
130	/*
131	Turn counts to milliseconds
132	*/
133	network_timeout = short_network_timeout;
134	network_timeout *= MSEC2MS_PER_SECOND;
135	
136	}
137	/*
138	Check reboot
139	*/
140	check_reboot();
141	
142	/*
143	if we are calibrating DMC, set calibrate_led
144	*/
145	if (calibrate_led == 1) {
146	if (value == 0) {
147	clear_test_led();
148	value = 1;
149	} else {
150	set_test_led();
151	value = 0;
152	}
153	}
154	
155	/*
156	go thru conn structures checking for timeouts
157	*/
158	conn_table = table_of_conns(0);
159	for (users = 0, users != MAX_USERS, users++) {
160	conn = *conn_table++;
161	
162	if (conn == (CONNECTION *) NULL)
163	continue;
164	
165	CPU_DISABLE_INTERRUPTS;
166	
167	/*
168	close the connection
169	*/
170	if (conn->do_close == TRUE) {
171	if (close_connection_for_server(conn) == FAILURE) {
172	CPU_ENABLE_INTERRUPTS;
173	printf("Unable to close connection %d\n", conn->id);
174	CPU_DISABLE_INTERRUPTS;
175	
176	}
177	} else {
178	
179	/*
180	did we send the last part of the reply and is this
181	connection supposed to close
182	*/
183	if (conn->as_reading == FALSE && conn->as_closing == TRUE) {
184	conn->do_close = TRUE;
185	}
186	
187	/*
188	network timeout is set, and we are timing out
189	*/
190	if (network_timeout != 0 && conn != (CONNECTION *) NULL &&
191	conn->as_timing_out == TRUE) {
192	if (conn->time_to_live < (lm_tick * 5)) {
193	CPU_ENABLE_INTERRUPTS;
194	if (check_connection_for_life(conn) == FAILURE) {
195	printf("Can't check connection for life in housekeeping task user %d\n",
196	users);
197	
198	conn->time_to_live = (lm_tick * 5) + network_timeout;
199	if (conn->number_of_live_retries++ > MAX_TRIES) {
200	
201	/*
202	The host device is not responding to device
203	requests. This does not mean that the host is dead,
204	so just make a note of it and continue.
205	*/
206	conn->number_of_live_retries = 0;
207	
208	}
209	}
210	
211	if ((conn = table_of_conns(MAX_USERS)) == (CONNECTION *) NULL) {
212	CPU_ENABLE_INTERRUPTS;
213	continue;
214	
215	/*
216	close the connection
217	*/
218	if (conn->do_close == TRUE) {
219	if (close_connection_for_server(conn) == FAILURE) {
220	CPU_ENABLE_INTERRUPTS;
221	printf("Unable to close connection %d\n", conn->id);
222	CPU_DISABLE_INTERRUPTS;
223	continue;
224	
225	}
226	
227	/*
228	Did we send the 1 packet reply and is this connection supposed to
229	close?
230	*/
231	if (conn->as_closing == TRUE) {
232	conn->do_close = TRUE;
233	
234	CPU_ENABLE_INTERRUPTS;
235	
236	}
237	
238	
239	
240	void

1731

5,353,243

1732

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM tasks.lm1000/hkeeping_task.c	DATE 5/23/89	PAGE # 3/3
TIME 4:42:33 pm				
SOURCE TEXT				
LINE #				
241	set_test_led()			
242	{			
243	cpu_control_reg_struct *p_cpu_ctl_reg =			
244	(cpu_control_reg_struct *) CPU_CONTROL_REG,			
245	p_cpu_ctl_reg->not_test_led = not_LED_ON,			
246	}			
247	void			
248	clear_test_led()			
249	{			
250	cpu_control_reg_struct *p_cpu_ctl_reg =			
251	(cpu_control_reg_struct *) CPU_CONTROL_REG,			
252	p_cpu_ctl_reg->not_test_led = not_LED_OFF,			
253	}			
254				
255				
256				

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM	DATE	PAGE #
		tasks.lm1000/receive_task.c	5/23/89	
			TIME 4:42:34 pm	1/4
LINE #	SOURCE TEXT			
1	/* SCCL ID: receive_task.c rev 1.1, 4/24/89 at 07:55:23 */			
2	/* receive.c */			
3	/*			
4	*/			
5	#include "common.h"			
6	#include "fifo.h"			
7	#include "cpu.h"			
8	#include "task.h"			
9	#include "message.h"			
10	#include "network.h"			
11				
12	u_long lm_global_clock;			
13				
14	extern CONNECTION *table_of_conns[ ];			
15				
16				
17	void			
18	receive_task()			
19	{			
20	{			
21	u_short type;			
22	char str[MAX_MESSAGE];			
23				
24	u_long rta;			
25	u_char user;			
26				
27	CONNECTION *conn;			
28				
29	/*			
30	** Initialize socket, before doing anything else			
31	** sets up base conn.			
32	*/			
33	if (init_socket() == FAILURE) {			
34	/* FEXIT */			
35	}			
36				
37	lm_global_clock = 0;			
38	for (;;) {			
39	{			
40	rta = lm_choose_connection( user );			
41	if (rta != FAILURE)			
42	{			
43	conn = table_of_conns[ user ];			
44	if ( rta == PENDING )			
45	{			
46	if (lm_send_reply(conn) != SUCCESS)			
47	{			
48	printf("aaaaaaaaaaaaaa");			
49	/* FEXIT */			
50	}			
51	}			
52	else			
53	{			
54	/* FEXIT */			
55	}			
56	}			
57	else			
58	{			
59	while (lm_dequeue_message( &type, str ) != FAILURE) {			
60	if (type == ERROR_MSG)			
61	(void) printf("ERROR: %s\n", str);			
62	else (void) printf("WARNING: %s\n", str);			
63	}			
64	}			
65	}			
66	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/arp.c

DATE 5/23/89  
TIME 4:42:10 pm

PAGE #  
1/1

```

1  /* SCCS ID: arp.c rev 3.1, 4/24/89 at 07:46:29 */
2
3  /*
4   * This file contains the ARP request and response
5   * functions. For information on ARP and RARP see
6   * the DDM protocol handbook, volume 3, pp 3-615.
7   */
8
9  #include "common.h"
10 #include "arp.h"
11
12 /* external function declarations */
13 extern unsigned short lsnce_transmit();
14 extern char *memcpy();
15
16
17
18 /*
19  * This routine broadcasts an ARP request for a modeler.
20  * The receive task will receive the reply and handle
21  * appropriately
22  */
23
24 send_arp_request (enet_address, inet_address, dest_inet_address)
25     char *enet_address;
26     unsigned long inet_address;
27     unsigned long dest_inet_address;
28 {
29     static struct ethernet_arp_packet arp_request;
30     long error;
31
32     (void) memcpy ((char *) arp_request.destination, BROADCAST, ARP_HARDWARE_SIZE);
33     (void) memcpy ((char *) arp_request.source, enet_address, ARP_HARDWARE_SIZE);
34
35     arp_request.type = ARP_ARP;
36
37     arp_request.arp.arp_hardware_type = ARP_ETHERNET;
38     arp_request.arp.arp_protocol_type = ARP_IP;
39     arp_request.arp.arp_hardware_address_length = ARP_HARDWARE_SIZE;
40     arp_request.arp.arp_protocol_address_length = ARP_PROTOCOL_SIZE;
41     arp_request.arp.arp_opcode = ARP_REQUEST;
42     arp_request.arp.arp_source_protocol_address = inet_address;
43     arp_request.arp.arp_target_protocol_address = dest_inet_address;
44
45     (void) memcpy ((char *) arp_request.arp.arp_source_hardware_address, enet_address, ARP_HARDWARE_SIZE);
46     (void) memcpy ((char *) arp_request.arp.arp_target_hardware_address, NULL_ADDRESS, ARP_HARDWARE_SIZE);
47
48     (void) lsnce_transmit (&arp_request, (short) sizeof (arp_request), &error);
49
50     return;
51 }
52
53
54
55 /*
56  * This routine generates a response to an ARP request. It is
57  * currently unimplemented. The modeler will not respond to
58  * ARP requests, a host machine must publish an ARP response
59  * for a modeler. I wrote this routine simply because it was
60  * easy to do while I had my name in the DDM book. If we ever
61  * implement a full ARP, I hope I saved someone some time and
62  * effort, if not, oh well...
63  */
64
65 send_arp_reply (arp_reply, enet_address, inet_address, dest_enet_address, dest_inet_address)
66     struct ethernet_arp_packet *arp_reply;
67     char *enet_address;
68     unsigned long inet_address;
69     char *dest_enet_address;
70     unsigned long dest_inet_address;
71 {
72     long error;
73
74     (void) memcpy ((char *) arp_reply->destination, dest_enet_address, ARP_HARDWARE_SIZE);
75     (void) memcpy ((char *) arp_reply->source, enet_address, ARP_HARDWARE_SIZE);
76
77     arp_reply->type = ARP_ARP;
78
79     arp_reply->arp.arp_hardware_type = ARP_ETHERNET;
80     arp_reply->arp.arp_protocol_type = ARP_IP;
81     arp_reply->arp.arp_hardware_address_length = ARP_HARDWARE_SIZE;
82     arp_reply->arp.arp_protocol_address_length = ARP_PROTOCOL_SIZE;
83     arp_reply->arp.arp_opcode = ARP_REPLY;
84     arp_reply->arp.arp_source_protocol_address = inet_address;
85     arp_reply->arp.arp_target_protocol_address = dest_inet_address;
86
87     (void) memcpy ((char *) arp_reply->arp.arp_source_hardware_address, enet_address, ARP_HARDWARE_SIZE);
88     (void) memcpy ((char *) arp_reply->arp.arp_target_hardware_address, dest_enet_address, ARP_HARDWARE_SIZE);
89
90     (void) lsnce_transmit (arp_reply, (short) sizeof (arp_reply), &error);
91
92     return;
93 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/bus.c	DATE 5/23/89 TIME 4:42:10 pm	PAGE # 1/2
LINE #	SOURCE TEXT			
1	/* SCCS ID: bus.c rev 3.1, 4/24/89 at 07:46:32 */			
2	/*			
3	** bus.c			
4	** bus error handler.			
5	** It is more complicated than it needs to be,			
6	** we need to use this to determine if a PAC			
7	** is present...			
8	*/			
9	#include "common.h"			
10	#include "cpu.h"			
11	#include "cpu_vec.h"			
12	#include "mod_err.h"			
13	#include "nvaram.h"			
14	#include "lm_rd_wr.h"			
15	#include "bus.h"			
16	static char buf[ 200 ];			
17	extern bus_isr();			
18	u_long setup_ivt_bus();			
19	extern u_short lm_nvaram_access();			
20	extern void reset_cpu();			
21	void bus_error_isr();			
22	void output_routine();			
23	u_short address_in_map();			
24	static u_char *ignore_low_address;			
25	static u_char *ignore_hi_address;			
26	u_char ignore_mem_error;			
27	static u_char mem_error;			
28	static char debug_bus = 1;			
29	/*			
30	** Bus error handler			
31	*/			
32	void bus_error( )			
33	{			
34	extern STACK_FRAME *bus_error_address;			
35	STACK_FRAME *stack_frame = bus_error_address;			
36	u_long err;			
37	if( address_in_map( stack_frame->address ) == SUCCESS)			
38	{			
39	if( ignore_mem_error == TRUE )			
40	{			
41	if( ignore_low_address >= stack_frame->address &&			
42	ignore_hi_address <= stack_frame->address )			
43	{			
44	if( stack_frame->spec_status_word & 0x0100)			
45	{			
46	stack_frame->spec_status_word  = 0x0100;			
47	mem_error = TRUE;			
48	return;			
49	}			
50	}			
51	}			
52	if( lm_nvaram_access( (char *) stack_frame->address, BUS_ADDR, (u_long) sizeof BUS_ADDR, MEMORY_WRITE, &err ) == FAILURE)			
53	{			
54	reset_cpu( SUICIDE );			
55	}			
56	if( lm_nvaram_access( (char *) stack_frame->spec_status_word, SSW_REG, (u_long) sizeof SSW_REG, MEMORY_WRITE, &err ) == FAILURE)			
57	{			
58	reset_cpu( SUICIDE );			
59	}			
60	if( lm_nvaram_access( (char *) stack_frame->vector, ERROR_VECTOR, (u_long) sizeof ERROR_VECTOR, MEMORY_WRITE, &err ) == FAILURE)			
61	{			
62	reset_cpu( SUICIDE );			
63	}			
64	}			
65	/* print out screen			
66	*/			
67	if( debug_bus == 1 )			
68	{			
69	sprintf( buf, "\nInterrupt priority level %0x", (stack_frame->sr & 0x700) >> 8 );			
70	output_routine( buf );			
71	sprintf( buf, "\nProgram Counter %0x", stack_frame->pc );			
72	output_routine( buf );			
73	sprintf( buf, "\nFailed at address %0x", stack_frame->address );			
74	output_routine( buf );			
75	if( stack_frame->spec_status_word & 0x100)			
76	{			
77	sprintf( buf, "\nData Access" );			
78	output_routine( buf );			
79	if( stack_frame->spec_status_word & 0x80)			
80	{			
81	sprintf( buf, "Read Modify write cycle " );			
82	output_routine( buf );			
83	}			
84	if( stack_frame->spec_status_word & 0x10)			
85	{			
86	sprintf( buf, "Byte " );			
87	output_routine( buf );			
88	}			
89	if( stack_frame->spec_status_word & 0x20)			
90	{			
91	sprintf( buf, "Word " );			
92	output_routine( buf );			
93	}			
94	if( ( ( stack_frame->spec_status_word & 0x30 )			
95	{			
96	sprintf( buf, "Long " );			
97	output_routine( buf );			
98	}			
99	}			
100	if( stack_frame->spec_status_word & 0x40)			
101	{			
102	sprintf( buf, "Read cycle " );			
103	output_routine( buf );			
104	}			
105	else			
106	{			
107	sprintf( buf, "Write cycle " );			
108	output_routine( buf );			
109	}			
110	}			
111	if( stack_frame->spec_status_word & 0x1000)			
112	{			
113	sprintf( buf, "\nInstruction Access " );			
114	output_routine( buf );			
115	}			
116	}			
117	/*			
118	** we crashed			
119	if( address_in_map( stack_frame->address ) == FAILURE)			
120	{			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/bus.c	DATE 5/23/89	PAGE # 2/3
LINE #	SOURCE TEXT			
121	reset_cpu( BUS_ERROR );			
122	}			
123				
124	#define ALWAYS_MEMORY 0			
125	#define NEVER_MEMORY 1			
126	#define OPTIONAL_MEMORY 2			
127	#define MAP_SIZE 32			
128	static u_long map[MAP_SIZE][2] = {			
129	0x00000000, ALWAYS_MEMORY,			
130	0x00000000, OPTIONAL_MEMORY,			
131	0x10000000, NEVER_MEMORY,			
132	0x10000000, OPTIONAL_MEMORY,			
133	0x10000000, ALWAYS_MEMORY,			
134	0x10000000, NEVER_MEMORY,			
135	0x20000000, NEVER_MEMORY,			
136	0x20000000, OPTIONAL_MEMORY,			
137	0x20000000, NEVER_MEMORY,			
138	0x20000000, OPTIONAL_MEMORY,			
139	0x30000000, NEVER_MEMORY,			
140	0x30000000, OPTIONAL_MEMORY,			
141	0x30000000, NEVER_MEMORY,			
142	0x30000000, OPTIONAL_MEMORY,			
143	0x40000000, NEVER_MEMORY,			
144	0x40000000, OPTIONAL_MEMORY,			
145	0x40000000, NEVER_MEMORY,			
146	0x40000000, OPTIONAL_MEMORY,			
147	0x50000000, NEVER_MEMORY,			
148	0x50000000, OPTIONAL_MEMORY,			
149	0x50000000, NEVER_MEMORY,			
150	0x50000000, OPTIONAL_MEMORY,			
151	0x60000000, NEVER_MEMORY,			
152	0x60000000, OPTIONAL_MEMORY,			
153	0x60000000, NEVER_MEMORY,			
154	0x60000000, OPTIONAL_MEMORY,			
155	0x70000000, NEVER_MEMORY,			
156	0x70000000, OPTIONAL_MEMORY,			
157	0x70000000, NEVER_MEMORY,			
158	0x70000000, OPTIONAL_MEMORY,			
159	0x80000000, NEVER_MEMORY,			
160	0x80000000, NEVER_MEMORY,			
161	};			
162	u_short			
163	address_in_map( address )			
164	u_char *address;			
165	{			
166	register u_char i;			
167	/*			
168	** Go thru above table to determine what to do..			
169	*/			
170	for( i = 0; i <= MAP_SIZE; ++i)			
171	{			
172	if( map[ i ][ 0 ] == (u_long) address )			
173	{			
174	if( map[ i ][ 1 ] == NEVER_MEMORY )			
175	return( FAILURE );			
176	return( SUCCESS );			
177	}			
178	}			
179	return( FAILURE );			
180	}			
181	/*			
182	** address for which errors are to be ignored			
183	*/			
184	void lm_ignore_bus_error( addr_lo, addr_hi )			
185	u_char *addr_lo, *addr_hi;			
186	{			
187	ignore_low_address = addr_lo;			
188	ignore_hi_address = addr_hi;			
189	ignore_mem_error = TRUE;			
190	mem_error = FALSE;			
191	}			
192	/*			
193	** did we get a bus error			
194	*/			
195	lm_check_bus_error( )			
196	{			
197	return( mem_error );			
198	}			
199	/*			
200	** deal with bus error correctly, and do not ignore any errors			
201	*/			
202	void lm_enable_bus_error()			
203	{			
204	ignore_mem_error = FALSE;			
205	}			
206	u_short			
207	lm_read_probe( addr )			
208	register u_long *addr;			
209	{			
210	cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
211	u_long junk;			
212	temp = setup_ivt_bus( (u_long) bus_iar );			
213	lm_ignore_bus_error((u_char *)addr, (u_char *)addr);			
214	/*			
215	** read.			
216	*/			
217	junk = *addr;			
218	lm_enable_bus_error();			
219	(void) setup_ivt_bus( temp );			
220	if( !lm_check_bus_error() )			
221	return( FAILURE );			
222	return( SUCCESS );			
223	}			
224	u_short			
225	lm_write_probe( addr, value )			
226	register u_long *addr, value;			
227	{			
228	u_long temp;			
229	cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
230	temp = setup_ivt_bus( (u_long) bus_iar );			
231	lm_ignore_bus_error((u_char *)addr, (u_char *)addr);			
232	/*			
233	** write.			
234	*/			
235	*addr = value;			
236	lm_enable_bus_error();			
237	}			
238				
239				
240				

1741

5,353,243

1742

Copyright 1989  
Logic Modeling SystemsSOURCE PROGRAM  
os/bus.cDATE 5/23/89  
TIME 4:42:10 pmPAGE #  
3/4

LINE #	SOURCE TEXT
241	(void) setup_ivt_bus( temp );
242	if( ! check_bus_error() == TRUE )
243	return( FAILURE );
244	return( SUCCESS );
245	}
246	static
247	u_long setup_ivt_bus( bus_iar )
248	u_long bus_iar;
249	{
250	unsigned long *ivt = (unsigned long *)VEC_BASE_ADDRESS;
251	u_long temp;
252	/*
253	/* The bus handler
254	*/
255	temp = *(ivt + 2);
256	*(ivt + 2) = (unsigned long) bus_iar;
257	return( temp );
258	}
259	
260	void output_routine( buf )
261	char *buf;
262	{
263	char *bufptr = buf;
264	while( *bufptr )
265	{
266	Bus_error_tx( *bufptr++ );
267	}
268	}
269	

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/duart.c

DATE 5/23/89  
TIME 4:42:10 pm

PAGE #  
1/5

```

1  /* SCSS_ID: duart.c rev 3.1, 4/24/89 at 07:46:36 */
2  /*
3  /* Duart.c
4  /*
5  #include "cpu.h"
6  #include "duart.h"
7
8  /*
9  /* have UARTA & UARTB been initialized
10 /* 0 = no, 1 = yes.
11 /*
12 extern INTERRUPT_REG Wart_mask;
13
14 int lm_uarta_baud; /* current uarta baud rate */
15 int lm_uartb_baud; /* current uartb baud rate */
16
17 /*
18 /* Varta_init()
19 /* Initialize channel A of the DUART.
20 /*
21 Varta_init( baud )
22 unsigned int baud;
23 {
24     MODE_REG_1 mode_reg1, *p_mode_reg1 = (MODE_REG_1 *) (CPU_DUART + MODE_REG1_A);
25     MODE_REG_2 mode_reg2, *p_mode_reg2 = (MODE_REG_2 *) (CPU_DUART + MODE_REG2_A);
26     CMD_REG cmd_reg, *p_cmd_reg = (CMD_REG *) (CPU_DUART + CMD_REG_A);
27     ACR acr, *p_acr = (ACR *) (CPU_DUART + AUX_CNTL_REG);
28
29     /*
30     /* reset everything
31     /* through command register
32     /*
33     /* disable tx & rx
34     /*
35     cmd_reg.zero = 0;
36     cmd_reg.command = NOP;
37     cmd_reg.dis_tx = DUART_DISABLE;
38     cmd_reg.enb_tx = DUART_DISABLE;
39     cmd_reg.dis_rx = DUART_DISABLE;
40     cmd_reg.enb_rx = DUART_DISABLE;
41     *p_cmd_reg = cmd_reg;
42     /*
43     /* reset channel
44     /*
45     cmd_reg.command = RESET_CH;
46     *p_cmd_reg = cmd_reg;
47
48     /*
49     /* reset error
50     /*
51     cmd_reg.command = RESET_ERR;
52     *p_cmd_reg = cmd_reg;
53     /*
54     /* reset rcvr
55     /*
56     cmd_reg.command = RESET_RCVR;
57     *p_cmd_reg = cmd_reg;
58     /*
59     /* reset tx
60     /*
61     cmd_reg.command = RESET_TX;
62     *p_cmd_reg = cmd_reg;
63     /*
64     /* reset mode register
65     /*
66     cmd_reg.command = RESET_MODE_REG;
67     *p_cmd_reg = cmd_reg;
68
69     /* set up mode reg 1
70     /* char mode, 8 bits/char & no parity
71     /*
72     mode_reg1.tx_rts = DUART_DISABLE;
73     mode_reg1.tx_int = RX_INT_RDY;
74     mode_reg1.error_mode = ERR_MODE_CHAR;
75     mode_reg1.parity = PARITY_NONE;
76     mode_reg1.bits_per_char = BITS_PER_CHAR_8;
77     *p_mode_reg1 = mode_reg1;
78     /*
79     /* set up mode reg 2
80     /*
81     mode_reg2.chas_mode = CH_MODE_NORMAL;
82     mode_reg2.tx_rts = DUART_DISABLE;
83     mode_reg2.cts_enable = DUART_DISABLE;
84     mode_reg2.stop_len = STOP_BITS_2;
85     *p_mode_reg2 = mode_reg2;
86
87     /* clock select
88     /*
89     Varta_baud( baud );
90     /*
91     /* set up aux. control reg
92     /*
93     acr.zero = 0;
94     *p_acr = acr;
95     /*
96     /* set up int. mask reg
97     /* this controls ints. for both UART's
98     /* we must take this into consideration.
99     /*
100    Varta_inr();
101    /*
102    /* enable tx & rx operations
103    /*
104    cmd_reg.command = NOP;
105    cmd_reg.dis_tx = DUART_DISABLE;
106    cmd_reg.enb_tx = DUART_ENABLE;
107    cmd_reg.dis_rx = DUART_DISABLE;
108    cmd_reg.enb_rx = DUART_ENABLE;
109    *p_cmd_reg = cmd_reg;
110
111    Varta_baud( baud )
112    {
113        CLK_SEL clk_sel, *p_clk_sel = (CLK_SEL *) (CPU_DUART + CLK_SEL_REG_A);
114
115        lm_uarta_baud = baud & 0xf;
116    }
117
118
119
120

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/duart.c	DATE 5/23/89 TIME 4:42:10 pm	PAGE # 2/6
LINE #	SOURCE TEXT			
121	/*			
122	** clock select			
123	/*			
124	clk_sela.RX_clk = baud;			
125	clk_sela.TX_clk = baud;			
126	*p_clk_sela = clk_sela;			
127	}			
128	/*			
129	uartb_lmr()			
130	{			
131	INTERRUPT_REG  interrupt_reg, *p_interrupt_reg = (INTERRUPT_REG *) (CPU_DUART + INT_MASK_REG);			
132	/*			
133	** set up int. mask reg			
134	** this controls ints. for both UART's			
135	** we must take this into consideration.			
136	/*			
137	interrupt_reg = uartb_mask;			
138	interrupt_reg.in_port = DUART_DISABLE;			
139	interrupt_reg.delta_brk_b = DUART_DISABLE;			
140	interrupt_reg.counter = DUART_DISABLE;			
141	interrupt_reg.delta_brk_a = DUART_DISABLE;			
142	interrupt_reg.rx_rdy_a = DUART_ENABLE;			
143	interrupt_reg.tx_rdy_a = DUART_ENABLE;			
144	/*			
145	** set interrupt reg. and save in mask this is a write only reg.			
146	/*			
147	uartb_mask = interrupt_reg;			
148	*p_interrupt_reg = interrupt_reg;			
149	}			
150	/*			
151	uartb_init( baud )			
152	{			
153	unassigned int baud;			
154	/*			
155	MODE_REG_1      mode_reg1b, *p_mode_reg1b = (MODE_REG_1 *) (CPU_DUART + MODE_REG1_B);			
156	MODE_REG_2      mode_reg2b, *p_mode_reg2b = (MODE_REG_2 *) (CPU_DUART + MODE_REG2_B);			
157	CLK_SEL          clk_selb, *p_clk_selb  = (CLK_SEL *)  (CPU_DUART + CLK_SEL_REG_B);			
158	CMD_REG          cmd_regb, *p_cmd_regb  = (CMD_REG *)  (CPU_DUART + CMD_REG_B);			
159	/*			
160	in_uartb_baud = baud & 0xf;			
161	/*			
162	/* reset everything			
163	/* through command register			
164	/*			
165	/* disable tx & rx			
166	/*			
167	cmd_regb.iaro = 0;			
168	cmd_regb.command = NOP;			
169	cmd_regb.dis_tx = DUART_ENABLE;			
170	cmd_regb.ena_tx = DUART_DISABLE;			
171	cmd_regb.dis_rx = DUART_ENABLE;			
172	cmd_regb.ena_rx = DUART_DISABLE;			
173	*p_cmd_regb = cmd_regb;			
174	/* reset channel			
175	/*			
176	cmd_regb.command = RESET_CH;			
177	*p_cmd_regb = cmd_regb;			
178	/*			
179	/* reset error			
180	/*			
181	cmd_regb.command = RESET_ERR;			
182	*p_cmd_regb = cmd_regb;			
183	/*			
184	/* reset rxvr			
185	/*			
186	cmd_regb.command = RESET_RCVR;			
187	*p_cmd_regb = cmd_regb;			
188	/*			
189	/* reset tx			
190	/*			
191	cmd_regb.command = RESET_TX;			
192	*p_cmd_regb = cmd_regb;			
193	/*			
194	/* reset mode register			
195	/*			
196	cmd_regb.command = RESET_MODE_REG;			
197	*p_cmd_regb = cmd_regb;			
198	/*			
199	/* set up mode reg 1			
200	/* char mode, 8 bits/char & no parity			
201	/*			
202	mode_reg1b.rx_rts = DUART_DISABLE;			
203	mode_reg1b.rx_int = RX_INT_RRDY;			
204	mode_reg1b.error_mode = ERR_MODE_CHAR;			
205	mode_reg1b.parity = PARITY_NONE;			
206	mode_reg1b.bits_per_char = BITS_PER_CHAR_8;			
207	*p_mode_reg1b = mode_reg1b;			
208	/*			
209	/* set up mode reg 2			
210	/*			
211	mode_reg2b.as_mode = CE_MODE_NORMAL;			
212	mode_reg2b.rx_rts = DUART_DISABLE;			
213	mode_reg2b.cts_enable = DUART_DISABLE;			
214	mode_reg2b.stop_lee = STOP_BITS_2;			
215	*p_mode_reg2b = mode_reg2b;			
216	/*			
217	/* clock select			
218	/*			
219	uartb_baud(baud);			
220	/*			
221	/* set up int. mask reg			
222	/* this controls ints. for both UART's			
223	/* we must take this into consideration.			
224	/*			
225	uartb_lmr();			
226	/*			
227	/* enable tx & rx operations			
228	/*			
229	cmd_regb.command = NOP;			
230	cmd_regb.dis_tx = DUART_DISABLE;			
231	cmd_regb.ena_tx = DUART_ENABLE;			
232	cmd_regb.dis_rx = DUART_DISABLE;			
233	cmd_regb.ena_rx = DUART_ENABLE;			
234	*p_cmd_regb = cmd_regb;			
235	/*			
240	/*			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/duart.c	DATE 5/23/89	PAGE # 3/7
LINE #	SOURCE TEXT			
241	}			
242	uart_baud( baud )			
243	{			
244	unsigned int baud;			
245	{			
246	CLK_SEL clk_selb, *p_clk_selb = (CLK_SEL *) (CPU_DUART + CLK_SEL_REG_B),			
247	lm_uart_baud = baud & 0xf,			
248	/*			
249	** clock select			
250	*/			
251	clk_selb.RX_clk = baud;			
252	clk_selb.TX_clk = baud;			
253	*p_clk_selb = clk_selb;			
254	}			
255	}			
256	uart_isr()			
257	{			
258	INTERRUPT_REG  interrupt_reg, *p_interrupt_reg = (INTERRUPT_REG *) (CPU_DUART + INT_MASK_REG),			
259	/*			
260	** set up int. mask reg			
261	** this controls ints. for both UART's			
262	** we must take this into consideration.			
263	*/			
264	interrupt_reg = uart_mask;			
265	interrupt_reg.RX_rdy_b = DUART_ENABLE ;			
266	interrupt_reg.TX_rdy_b = DUART_ENABLE ;			
267	/*			
268	** set interrupt reg, and save its mask this is a write only reg.			
269	*/			
270	uart_mask = interrupt_reg;			
271	*p_interrupt_reg = interrupt_reg;			
272	}			
273	/* UART BREAK RELATED FUNCTIONS */			
274	uart_rx_brk()			
275	{			
276	int nvaubaud;			
277	lm_get_nvaubaud(&nvaubaud);			
278	if ((nvaubaud & 0xf0) != 0x10)			
279	return;			
280	uart_baud(get_next_baud(lm_uart_baud));			
281	}			
282	uart_tx_brk()			
283	{			
284	int nvaubaud;			
285	lm_get_nvaubaud(&nvaubaud);			
286	if ((nvaubaud & 0xf0) != 0x10)			
287	return;			
288	uart_baud(get_next_baud(lm_uart_baud));			
289	}			
290	static			
291	get_next_baud(baud)			
292	{			
293	switch (baud) {			
294	default:			
295	case BAUD_1200:			
296	return BAUD_2400;			
297	case BAUD_2400:			
298	return BAUD_9600;			
299	case BAUD_9600:			
300	return BAUD_1200;			
301	}			
302	}			
303	}			
304	}			
305	}			
306	}			
307	}			
308	}			
309	}			
310	}			
311	}			
312	}			
313	}			
314	}			
315	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/eprom.c	DATE 5/23/89 TIME 4:42:11 pm	PAGE # 1/8
LINE #	SOURCE TEXT			
1	/* SCES_ID: eprom.c rev 3.1, 4/24/89 at 07:46:33 */			
2	/*			
3	** EProm Access routines			
4	*/			
5	#include "common.h"			
6	#include "mod_def.h"			
7	#include "magic.h"			
8	#include "pel.h"			
9	#include "eprom.h"			
10	#include "ls_rd_wr.h"			
11	#define REG_ADDRESS 8			
12	#define REG_DATA 0xa			
13				
14	extern u_char diag_dab;			
15	u_char debug_eprom = 0;			
16	static unsigned short reset[] = {			
17				
18	SET_CLK_0,			
19	SET_CS_0,			
20	SET_CS_1,			
21	SET_DATA_IN_LOW,			
22	TOG_CLK,			
23	NOP,			
24	NOP,			
25	NOP,			
26	NOP,			
27	NOP,			
28	NOP,			
29	NOP,			
30	NOP,			
31	NOP,			
32	NOP,			
33	NOP,			
34	NOP,			
35	NOP,			
36	NOP,			
37	NOP,			
38	NOP,			
39	NOP,			
40	NOP,			
41	NOP,			
42	NOP,			
43	NOP,			
44	NOP,			
45	NOP,			
46	NOP,			
47	NOP,			
48	NOP,			
49	NO_TOG_CLK,			
50	SET_CS_0,			
51	SET_CLK_0,			
52	DONE ;			
53	static unsigned short read_command[] = {			
54				
55	SET_CLK_0,			
56	SET_CS_1,			
57	TOG_CLK,			
58	LITERAL + 1,			
59	START_OPCODE,			
60	LITERAL + 2,			
61	EEPROM_READ,			
62	LITERAL + 6,			
63	0,			
64	SET_DATA_IN_LOW, /* THIS IS WHERE THE ADDRESS GOES */			
65	NOP,			
66	READ_DATA,			
67	SET_CS_0,			
68	SET_CS_0,			
69	SET_CLK_0,			
70	NO_TOG_CLK,			
71	DONE ;			
72	static unsigned short write_command[] = {			
73				
74	SET_CLK_0,			
75	SET_CS_1,			
76	TOG_CLK,			
77	LITERAL + 1,			
78	START_OPCODE,			
79	LITERAL + 2,			
80	EEPROM_WRITE,			
81	LITERAL + 6,			
82	0, /* THIS IS WHERE THE ADDRESS GOES */			
83	LITERAL + 16, /* THIS IS WHERE THE DATA GOES */			
84	0,			
85	SET_CS_0,			
86	SET_CS_0,			
87	/* WAIT_OUT_LOW,*/			
88	WAIT_OUT_HIGH,			
89	SET_CS_0,			
90	NO_TOG_CLK,			
91	DONE ;			
92	static unsigned short erase_command[] = {			
93				
94	SET_CLK_0,			
95	SET_CS_1,			
96	TOG_CLK,			
97	LITERAL + 1,			
98	START_OPCODE,			
99	LITERAL + 2,			
100	EEPROM_ERASE,			
101	LITERAL + 6,			
102	0, /* THIS IS WHERE THE ADDRESS GOES */			
103	SET_DATA_IN_LOW,			
104	NOP,			
105	NOP,			
106	SET_CS_0,			
107	SET_CS_0,			
108	SET_CS_1,			
109	/* WAIT_OUT_LOW,*/			
110	WAIT_OUT_HIGH,			
111	SET_CS_0,			
112	NO_TOG_CLK,			
113	DONE ;			
114	static unsigned short write_enable[] = {			
115				
116	SET_CLK_0,			
117	SET_CS_1,			
118	TOG_CLK,			
119	LITERAL + 1,			
120	START_OPCODE,			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/eprom.c	DATE 5/23/89	PAGE # 2/9
LINE #	SOURCE TEXT			
121	LITERAL + 2,			
122	EEPROM_ALL,			
123	LITERAL + 6,			
124	EEPROM_END,			
125	SET_CS_0,			
126	NOP,			
127	NOP,			
128	NOP,			
129	NO_TOC_CLK,			
130	DONE ;			
131				
132	static unsigned short write_disable[] = {			
133	SET_CLK_0,			
134	SET_CS_1,			
135	TOC_CLK,			
136	LITERAL + 1,			
137	START_OPCODE,			
138	LITERAL + 2,			
139	EEPROM_ALL,			
140	LITERAL + 6,			
141	EEPROM_END,			
142	SET_CS_0,			
143	NOP,			
144	NOP,			
145	NOP,			
146	NO_TOC_CLK,			
147	DONE ;			
148				
149	static unsigned short write_all_command[] = {			
150	SET_CLK_0,			
151	SET_CS_1,			
152	TOC_CLK,			
153	LITERAL + 1,			
154	START_OPCODE,			
155	LITERAL + 2,			
156	EEPROM_ALL,			
157	LITERAL + 6,			
158	EEPROM_END,			
159	LITERAL + 16,			
160	0 /* THIS IS WHERE THE DATA DOES */			
161	SET_CS_0,			
162	SET_CS_0,			
163	SET_CS_1,			
164	/* WAIT OUT LOW, */			
165	WAIT_OUT_HIGH,			
166	SET_CS_0,			
167	NO_TOC_CLK,			
168	DONE ;			
169				
170				
171	static unsigned short erase_all_command[] = {			
172	SET_CLK_0,			
173	SET_CS_1,			
174	TOC_CLK,			
175	LITERAL + 1,			
176	START_OPCODE,			
177	LITERAL + 2,			
178	EEPROM_ALL,			
179	LITERAL + 6,			
180	EEPROM_END,			
181	NOP,			
182	NOP,			
183	SET_CS_0,			
184	SET_CS_0,			
185	SET_CS_1,			
186	/* WAIT OUT LOW, */			
187	WAIT_OUT_HIGH,			
188	SET_CS_0,			
189	NO_TOC_CLK,			
190	DONE			
191	};			
192				
193	DAB EEPROM dab_a2,			
194	extern u_short is_a2prom_access();			
195	u_short access_eprom();			
196	u_short write_eprom(), read_eprom();			
197	void disable_eprom(), enable_eprom(), erase_all_eprom();			
198				
199	static void			
200	summary_copy(a1, a2, count)			
201	register u_long *a1;			
202	register u_long *a2;			
203	register short count;			
204	{			
205	register short remain = count + 3;			
206	{			
207	/*			
208	* divide by 4,			
209	* how many long moves we need to perform			
210	*/			
211	count >>= 2;			
212	while(--count >= 0) {			
213	*a1++ = *a2++;			
214	}			
215	/*			
216	* clean up, by moving remaining bytes			
217	*/			
218	while(--remain >= 0) {			
219	*(u_char *)a1++ = *(u_char *)a2++;			
220	}			
221	}			
222				
223	#ifdef DEMOC			
224	static u_long error = 0;			
225	main()			
226	{			
227	u_short eedata;			
228	{			
229	u_char ee_number = 2;			
230	erase_command[ REG_ADDRESS ] = 0x22;			
231	write_command[ REG_DATA ] = 0x44;			
232	write_command[ REG_ADDRESS ] = 0x22;			
233	(void) access_eprom(ee_number, &erase_command[ 0 ], &edata);			
234	printf("erase");			
235	(void) access_eprom(ee_number, &write_command[ 0 ], &edata);			
236	{			
237	test_eprom(ee_number)			
238	u_long ee_number;			
239	{			
240	{			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/eeeprom.c

DATE 5/23/89  
TIME 4:42:11 pm

PAGE #  
3/10

## SOURCE TEXT

```

241 printf("test_eeeprom\n");
242 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
243 {
244     printf("INVALID eeprom\n");
245     if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_INIT, &error) == FAILURE)
246     {
247         printf("ERROR INITIALIZING EEPROM\n");
248     }
249 }
250 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
251 {
252     printf("ERROR RE_VALIDATING EEPROM\n");
253 }
254 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_READ, &error) == FAILURE)
255 {
256     printf("ERROR READING\n");
257 }
258 /*
259 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_WRITE, &error) == FAILURE)
260 {
261     printf("ERROR WRITING\n");
262 }
263 */
264 if( !m_eeeprom_access( (char *)dab_e2, (u_long) 0, ee_number, (u_long) sizeof( dab_e2 ), MEMORY_VALIDATE, &error) == FAILURE)
265 {
266     printf("ERRORSXXXXXXXXXXXX\n");
267 }
268 printf("SUCCESSFULLY TESTED EEPROM\n");
269 }
270 }
271 #endif
272 void
273 enable_eeeprom( ee_number )
274 u_char ee_number;
275 {
276     u_short eedata;
277     (void)access_eeeprom(ee_number,&write_enable[ 0 ], &eedata);
278 }
279 void
280 erase_all_eeeprom( ee_number )
281 u_char ee_number;
282 {
283     u_short eedata;
284     enable_eeeprom( ee_number );
285     (void)access_eeeprom(ee_number,&erase_all_command[ 0 ], &eedata);
286     disable_eeeprom( ee_number );
287 }
288 void
289 disable_eeeprom( ee_number )
290 u_char ee_number;
291 {
292     u_short eedata;
293     (void)access_eeeprom(ee_number,&write_disable[ 0 ], &eedata);
294 }
295 u_short
296 write_eeeprom( ee_number, address, data )
297 u_char ee_number;
298 u_char address;
299 u_short data;
300 {
301     u_short w_cmd[ sizeof(write_command)], e_cmd[ sizeof(erase_command)];
302     u_short eedata;
303     u_short status;
304     (void)memory_copy( (u_long *)e_cmd, (u_long *)erase_command, (short)sizeof(erase_command));
305     (void)memory_copy( (u_long *)w_cmd, (u_long *)write_command, (short)sizeof(write_command));
306     e_cmd[ REG_ADDRESS ] = address;
307     w_cmd[ REG_ADDRESS ] = address;
308     w_cmd[ REG_DATA ] = data;
309     if( access_eeeprom( ee_number, &write_enable[ 0 ], &eedata) == FAILURE) return( FAILURE);
310     if( access_eeeprom( ee_number, &e_cmd[ 0 ], &eedata) == FAILURE) return( FAILURE);
311     if( access_eeeprom( ee_number, &w_cmd[ 0 ], &eedata) == FAILURE) return( FAILURE);
312     if( access_eeeprom( ee_number, &write_disable[ 0 ], &eedata) == FAILURE) return( FAILURE);
313     if( read_eeeprom( ee_number, address, &status ) != data)
314     {
315         if( debug_eeeprom == 1)
316             printf("error reading ee_number tx address tx expected data tx", ee_number, address, data);
317         return( FAILURE );
318     }
319     return( status );
320 }
321 u_short
322 read_eeeprom( ee_number, address, status )
323 u_short *status;
324 u_char ee_number;
325 u_char address;
326 {
327     u_short r_cmd[ sizeof(read_command)], eedata;
328     memory_copy( (u_long *)r_cmd, (u_long *)read_command,
329                 (short)sizeof(read_command));
330     r_cmd[ REG_ADDRESS ] = address;
331     *status = access_eeeprom(ee_number, &r_cmd[ 0 ], &eedata);
332     return( eedata );
333 }
334 #define ACCESS_TIMEOUT 1000
335 u_short
336 access_eeeprom( ee_number, array, data )
337 u_char ee_number;
338 unsigned short array[];
339 register unsigned short *data;
340 {
341     register PEL *pel_access_reg;
342     register unsigned short *parray = &array[0];
343     register unsigned char skip = FALSE;
344     register unsigned char count = 16, toggle = 0;
345     register int timeout = 0;
346     *data = 0;
347     pel_access_reg = (PEL *)
348         (pel_addr(((ee_number & 0x1f) >> 3), (ee_number & 7)));
349     /*
350     ** if EEDOUT is stuck low, return FAILURE.
351     */

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/eprom.c	DATE 5/23/89	PAGE # 4/11
LINE #	SOURCE TEXT			
361	if( read_out( pel_access_reg ) == 0 )			
362	{			
363	printf("ERROR in stack low\n");			
364	return( FAILURE );			
365	}			
366	while ( *pararray != DONE )			
367	{			
368	if ( timeout >= ACCESS_TIMEOUT ) break;			
369	/*			
370	skip for delay deb			
371	** otherwise, check active bit.			
372	*/			
373	if( diag_deb == 0 && pel_access_reg <> car.bit.active == 0 )			
374	{			
375	set_cs( pel_access_reg, 0 );			
376	set_clk( pel_access_reg, 0 );			
377	write_data( pel_access_reg, 0 );			
378	return( FAILURE );			
379	}			
380	if( toggle )			
381	{			
382	if( skip == FALSE )			
383	set_clk( pel_access_reg, 0 );			
384	skip = FALSE;			
385	}			
386	if( *pararray & LITERAL )			
387	{			
388	if( *pararray == LITERAL )			
389	{			
390	if( count != 0 )			
391	{			
392	if( count > (*pararray - LITERAL) )			
393	count = *pararray - LITERAL;			
394	count--;			
395	write_data( pel_access_reg,			
396	(*pararray + 1) >> count & 1 );			
397	}			
398	} else			
399	{			
400	count = 16;			
401	pararray++;			
402	pararray++;			
403	skip = TRUE;			
404	continue;			
405	}			
406	}			
407	}			
408	else			
409	{			
410	switch( *pararray++ )			
411	{			
412	case SET_CLK_0:			
413	set_clk( pel_access_reg, 0 );			
414	break;			
415	case SET_CLK_1:			
416	set_clk( pel_access_reg, 1 );			
417	break;			
418	case TOG_CLK:			
419	toggle = 1;			
420	break;			
421	case NO_TOG_CLK:			
422	toggle = 0;			
423	break;			
424	case SET_CS_0:			
425	set_cs( pel_access_reg, 0 );			
426	break;			
427	case SET_CS_1:			
428	set_cs( pel_access_reg, 1 );			
429	break;			
430	case READ_DATA:			
431	if( count-- )			
432	{			
433	pararray--;			
434	*data = (*data << 1)			
435	+ read_out( pel_access_reg );			
436	}			
437	} else			
438	count = 16;			
439	break;			
440	case WRITE_DATA:			
441	if( count-- )			
442	{			
443	pararray--;			
444	write_data( pel_access_reg,			
445	*data & ( 1 << (count - 1) );			
446	}			
447	} else			
448	count = 16;			
449	break;			
450	case WAIT_OUT_LOW:			
451	if( read_out( pel_access_reg ) == 1 ) {			
452	++timeout;			
453	pararray--;			
454	}			
455	break;			
456	case WAIT_OUT_HIGH:			
457	if( read_out( pel_access_reg ) == 0 ) {			
458	++timeout;			
459	pararray--;			
460	}			
461	break;			
462	case WAIT_IN_LOW:			
463	if( read_in( pel_access_reg ) == 1 ) {			
464	++timeout;			
465	pararray--;			
466	}			
467	break;			
468	case WAIT_IN_HIGH:			
469	if( read_in( pel_access_reg ) == 0 ) {			
470	++timeout;			
471	pararray--;			
472	}			
473	break;			
474	case SET_DATA_IN_LOW:			
475	write_data( pel_access_reg, 0 );			
476	skip = TRUE;			
477	continue;			
478	}			
479	case NOP:			
480	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/eprom.c	DATE 5/23/89	PAGE # 5/12
LINE #		SOURCE TEXT		
481		break;		
482		}		
483		if( toggle )		
484		{		
485		set_clk( pel_access_reg, 1 );		
486		{		
487		}		
488		}		
489		return (timeout < ACCESS_TIMEOUT)? SUCCESS: FAILURE;		
490		{		
491		}		
492		set_clk( pel_access_reg, 1 )		
493		PEL "pel_access_reg,		
494		u_char i;		
495		{		
496		#ifdef DEBUG		
497		printf( "\nCLK=td", i );		
498		#else		
499		#endif		
500		if( i )		
501		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit )   0x40 );		
502		else		
503		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit ) & 0xbf );		
504		#else		
505		pel_access_reg->car.bit.eeprom_clk = 1;		
506		#endif		
507		#endif		
508		{		
509		set_cs( pel_access_reg, 1 )		
510		PEL "pel_access_reg,		
511		u_char i;		
512		{		
513		#ifdef DEBUG		
514		printf( "\nCS=td", i );		
515		#else		
516		#endif		
517		if( i )		
518		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit )   0x80 );		
519		else		
520		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit ) & 0x7f );		
521		#else		
522		pel_access_reg->car.bit.eeprom_sel = 1;		
523		#endif		
524		#endif		
525		{		
526		read_out( pel_access_reg )		
527		PEL "pel_access_reg,		
528		{		
529		#ifdef DEBUG		
530		printf( "\nRead out",		
531		if( getchar() == '1' )		
532		{		
533		return( 1 );		
534		}		
535		return 0;		
536		#else		
537		#endif		
538		if( read_loc_short( pel_access_reg->car.bit ) & 0x100 )		
539		#else		
540		if( pel_access_reg->car.bit.eeprom_out == 1 )		
541		#endif		
542		{		
543		return( 1 );		
544		}		
545		return 0;		
546		#endif		
547		{		
548		read_in( pel_access_reg )		
549		PEL "pel_access_reg,		
550		{		
551		#ifdef DEBUG		
552		printf( "\nRead in",		
553		if( getchar() == '1' )		
554		{		
555		return( 1 );		
556		}		
557		return 0;		
558		#else		
559		#endif		
560		if( read_loc_short( pel_access_reg->car.bit ) & 0x20 )		
561		#else		
562		if( pel_access_reg->car.bit.eeprom_in == 1 )		
563		#endif		
564		{		
565		return( 1 );		
566		}		
567		return 0;		
568		#endif		
569		{		
570		write_data( pel_access_reg, 1 )		
571		PEL "pel_access_reg,		
572		unsigned char i;		
573		{		
574		#ifdef DEBUG		
575		printf( "\nwrite data tx", i );		
576		#else		
577		#endif		
578		if( i )		
579		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit )   0x20 );		
580		else		
581		write_loc_short( pel_access_reg->car.bit, read_loc_short( pel_access_reg->car.bit ) & 0xdf );		
582		#else		
583		pel_access_reg->car.bit.eeprom_in = 1;		
584		#endif		
585		#endif		
586		{		

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/fifo.c

DATE 5/23/89  
TIME 4:42:11 pm

PAGE #  
1/13

```

1  /* SCCS ID: fifo.c rev 3.1, 4/24/89 at 07:46:42 */
2
3  /*
4   * fifo.c
5   * fifo handling for CPU board.
6   */
7  #include "common.h"
8  #include "fifo.h"
9  #include "cpu.h"
10
11 /* The event flag
12 */
13 #define event_flag
14 #define fifo_group
15
16 /* fifos for the whole system
17 */
18 struct fifo_type fifo[ MAX_FIFOS + 1 ];
19
20 /* semaphores, and their IDs
21 */
22 #define fifo_0_id
23 #define semaphore_id[ MAX_FIFOS + 1 ]
24 #define semaphore_count[ MAX_FIFOS + 1 ]
25
26 /* headers for each fifo entry.
27 */
28 struct fifo_header_type fifo_header[ MAX_FIFO_ENTRIES ];
29
30 /*
31 * Initialize fifo's and the headers.
32 */
33 void fifo_initialize()
34 {
35     register int i;
36     unsigned int err;
37     register struct fifo_header_type *pfifo_header = &fifo_header[ 0 ];
38
39     /* set the fifo data structure to zero.
40     */
41     event_flag = 0;
42     bzero(fifo, sizeof(fifo));
43     bzero(fifo_header, sizeof(fifo_header));
44
45     /* The headers organized in a queue, and associated with fifo 0
46     */
47     for( i = 0; i <= MAX_FIFO_ENTRIES-1; i++, pfifo_header++ )
48         pfifo_header->next = pfifo_header + 1;
49
50     fifo[ HEADER_FIFO ].count = MAX_FIFO_ENTRIES;
51     fifo[ HEADER_FIFO ].head = &fifo_header[ 0 ];
52     fifo[ HEADER_FIFO ].tail = &fifo_header[ MAX_FIFO_ENTRIES - 1 ];
53
54     /*
55     * create semaphore 1 per fifo
56     */
57     for( i = 0; i <= MAX_FIFOS; i++ )
58     {
59         semaphore_id[ i ] = sc_create( 0, 1, &err );
60         semaphore_count[ i ] = 0;
61         if( err )
62         {
63             sys_out( "error creating SEMAPHORE\n" );
64             return( FAILURE );
65         }
66     }
67
68     /*
69     * This avoids access to the array semaphore_id
70     * allocation of ID should be linear
71     */
72     fifo_0_id = semaphore_id[ 0 ];
73     return( SUCCESS );
74 }
75
76 /*
77 * place a data buffer in a fifo(pfifo_entry->fifo_number)
78 */
79 void fifo_put( pfifo_entry )
80 {
81     register struct fifo_entry *pfifo_entry;
82     register struct fifo_header_type *pfifo_header;
83     register struct fifo_type *pfifo;
84     register char fifo_no = pfifo_entry->fifo_no;
85     register unsigned int *fifo_count = &semaphore_count[ fifo_no ];
86     int err = 0;
87
88     /*
89     * do a sanity check
90     */
91     if( fifo_no > MAX_FIFOS ) return( FAILURE );
92
93     /*
94     * grab a header
95     */
96     /*
97     * speed up access to fifo
98     */
99     pfifo = &fifo[ HEADER_FIFO ];
100
101     /*
102     * disable interrupts
103     */
104     CPU_DISABLE_INTERRUPTS;
105
106     /*
107     * are there any header buffers, for the ISR
108     */
109     if( pfifo_entry->task == ISR_TASK )
110     {
111         if( !pfifo->count )
112         {
113             CPU_ENABLE_INTERRUPTS;
114             return( FAILURE );
115         }
116     }
117     else
118     {
119         /*
120          * are there any header buffers, for the TASK
121          */
122         while( pfifo->count < MIN_TASK_ENTRIES )
123         {
124             event_flag |= ((unsigned int) 1 << HEADER_FIFO);
125         }
126     }
127 }

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/fifo.c	DATE 5/23/89	PAGE # 2/14
LINE #	SOURCE TEXT			
121	CPU_DISABLE_INTERRUPTS;			
122	sc_spend( semaphore_id( HEADER_FIFO ), 0, &err );			
123	if( err ) return( FAILURE );			
124	CPU_DISABLE_INTERRUPTS;			
125	}			
126	/*			
127	*/			
128	/* get pointer to first entry			
129	*/			
130	pfifo_header = pfifo->head;			
131	/*			
132	/* having retrieved, update pointers and count			
133	*/			
134	if( --pfifo->count )			
135	pfifo->head = pfifo_header->next;			
136	else			
137	pfifo->head = pfifo->tail = 0;			
138	/*			
139	/* point to appropriate fifo			
140	*/			
141	pfifo = &fifo( fifo_no );			
142	/*			
143	/* set up the header			
144	*/			
145	pfifo_header->user = pfifo_entry->user;			
146	pfifo_header->next = 0;			
147	pfifo_header->data = pfifo_entry->data;			
148	pfifo_header->task = pfifo_entry->task;			
149	/*			
150	/* add to the appropriate queue			
151	*/			
152	if( pfifo->count )			
153	pfifo->tail->next = pfifo_header;			
154	else			
155	pfifo->head = pfifo_header;			
156	pfifo->tail = pfifo_header;			
157	pfifo->count++;			
158	/*			
159	/* if ( *fifo_count )			
160	{			
161	/* semaphore_count( fifo_no )--;			
162	/* ( *fifo_count )--;			
163	/* system_call = 1;			
164	CPU_DISABLE_INTERRUPTS;			
165	/*			
166	/* should we try and wakeup someone			
167	/* to avoid an array access -> sc_spend( semaphore_id( fifo_no ), &err );			
168	/* use fifo_0_id.			
169	/*			
170	sc_spend( fifo_0_id + fifo_no, &err );			
171	/*			
172	if( err )			
173	return( FAILURE );			
174	CPU_DISABLE_INTERRUPTS;			
175	return( SUCCESS );			
176	}			
177	/*			
178	/* get a fifo entry			
179	*/			
180	fifo_get( pfifo_entry )			
181	struct fifo_entry *pfifo_entry;			
182	{			
183	register struct fifo_header_type *pfifo;			
184	register struct fifo_type *pfifo;			
185	register char fifo_no = pfifo_entry->fifo_no;			
186	int err = 0;			
187	int event = 0;			
188	/*			
189	/* if ( fifo_no > MAX_FIFOS ) return( FAILURE );			
190	/*			
191	/* set up pointer to appropriate fifo			
192	*/			
193	pfifo = &fifo( fifo_no );			
194	/*			
195	/* disable interrupts			
196	CPU_DISABLE_INTERRUPTS;			
197	/*			
198	/* while no entries stick around			
199	/*			
200	while( !pfifo->count )			
201	{			
202	semaphore_count( fifo_no )++;			
203	CPU_DISABLE_INTERRUPTS;			
204	/*			
205	/* To avoid an array access, use fifo_0_id			
206	/*			
207	sc_spend( fifo_0_id + fifo_no, 0, &err );			
208	if( err ) return( FAILURE );			
209	CPU_DISABLE_INTERRUPTS;			
210	/*			
211	/* get an entry			
212	*/			
213	pfifo->head = pfifo->tail;			
214	/*			
215	/* adjust pointers			
216	*/			
217	pfifo->head = pfifo->head->next;			
218	/*			
219	/* if no entry? tail = head = 0			
220	/*			
221	if( ! --pfifo->count )			
222	pfifo->tail = 0;			
223	/*			
224	/* prepare return data struct			
225	/*			
226	pfifo_entry->data = pfifo->data;			
227	pfifo_entry->user = pfifo->user;			
228	pfifo_entry->task = pfifo->task;			
229	/*			
230	/* return header to free queue			
231	/*			
232	pfifo = &fifo( HEADER_FIFO );			
233	/*			
234	/* adjust pointers and check			
235	/*			
236	/*			
237	/*			
238	/*			
239	/*			
240	/*			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/fifo.c	DATE 5/23/89 TIME 4:42:11 pm	PAGE # 3/15
LINE #	SOURCE TEXT			
241	/* if any one waiting for this			
242	*/			
243	if( pfifo->count )			
244	lifo->tail->next = pfifo;			
245	else			
246	pfifo->head = pfifo;			
247	/*			
248	/* the tail points to the new entry			
249	*/			
250	pfifo->tail = pfifo;			
251	/*			
252	/* update count			
253	*/			
254	pfifo->count++;			
255	/*			
256	/* is any one waiting for a header			
257	*/			
258	if( event_flag & ( 1 << HEADER_FIFO ) )			
259	event = 1;			
260	/*			
261	/* enable interrupts			
262	*/			
263	CPU_ENABLE_INTERRUPTS;			
264	/*			
265	/* post event to a flag			
266	*/			
267	if( event )			
268	{			
269	sc_spost( semaphore_id( HEADER_FIFO ), terr);			
270	}			
271	/*			
272	/* done			
273	*/			
274	if( err )			
275	return( FAILURE );			
276	return( SUCCESS );			
277	}			
278	/*			
279	/* return number of elements in a fifo			
280	*/			
281	fifo_inquiry( pfifo_entry )			
282	struct fifo_entry *pfifo_entry;			
283	{			
284	register struct fifo_type *pfifo;			
285	register char fifo_no = pfifo_entry-> fifo_no;			
286	{			
287	if( fifo_no > MAX_FIFOS ) return(0);			
288	/*			
289	/* set up pointer to appropriate fifo			
290	*/			
291	pfifo = &fifo( fifo_no );			
292	/*			
293	return pfifo->count;			
294	}			
295	}			
296	}			
297	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/id.c	DATE 5/23/89	PAGE # 1/16
LINE #		SOURCE TEXT		
1		/* SCCS_ID: id.c rev 3.1, 4/24/89 at 07:46:44 */		
2		#include "common.h"		
3		#include "ls_diags.h"		
4		#include "ls_afi.h"		
5		#include "id.h"		
6		/*		
7		* Compute and verify checksum on ID FROM		
8		* Pass pointer to first byte in ID FROM.		
9		* Subsequent ID_NUM_BYTES-1 bytes are 4 bytes apart.		
10		* Algorithm:		
11		*   initialize checksum to an arbitrary (but well-known) value		
12		*   for each byte (including checksum)		
13		*     circular left shift left checksum		
14		*     add data byte		
15		*     mask checksum to 3 bits		
16		* Returns computed checksum.		
17		*/		
18		int		
19		id_check(address)		
20		u_char *address;		
21		{		
22		register int checksum;		
23		register u_long byte_count;		
24		checksum= ID_CHECKSUM_INIT;		
25		for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)		
26		{		
27		checksum= (checksum << 1) + ((checksum & 0x80) >> 7);		
28		checksum+= *(address + 4 * byte_count);		
29		checksum+= 0xFF;		
30		}		
31		return(checksum);		
32		}		
33		/*		
34		* Load as ID FROM into a character buffer		
35		* address is the physical byte address of the first byte in the ID FROM.		
36		* buffer is the byte address of the first byte in a ID_NUM_BYTES buffer.		
37		* In practice, buffer is really an appropriate ID_FROM_XXX structure, and		
38		* a pointer to it is passed, cast to (u_char *).		
39		*/		
40		void		
41		id_load(address, buffer)		
42		u_char *address;		
43		u_char *buffer;		
44		{		
45		register int byte_count;		
46		for (byte_count= 0; byte_count < ID_NUM_BYTES; byte_count++)		
47		{		
48		*buffer= *address;		
49		buffer++;		
50		address+= 4;		
51		}		
52		}		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 1/17
LINE #		SOURCE TEXT		
1		/* SCCS ID: lance.c rev 3.2, 4/23/89 at 16:44:16 */		
2		#include "common.h"		
3		#include "cpu.h"		
4		#include "vrx.h"		
5		#include "lance.h"		
6		#include "id.h"		
7		#include "network.h"		
8		#include "mod_err.h"		
9		#include "svcs.h"		
10		#include "lm_xd_vr.h"		
11		#include "arp.h"		
12		#include "arp.h"		
13		#define SECOND 200		
14		#define TIMEOUT 33		
15		#define MAX_LANCE_ENABLE_POLLS 30		
16		#define MAX_INITIALIZED_DOME_POLLS 5		
17		/*		
18		* Memory for the LANCE initialization block.		
19		* init_block_mem() must be word aligned.		
20		*/		
21		char init_block_mem(alignof(INIT_BLOCK));		
22		u_long system_call;		
23		u_long rcv_lance_pkt_semaphore;		
24		static u_long alive_inet;		
25		u_char accept_arp_reply = 0;		
26		u_long modeler_inet;		
27		u_long modeler_inet_address;		
28		#ifdef BROKEN_HARDWARE		
29		char reinitialize_lance = 0;		
30		#endif BROKEN_HARDWARE		
31		extern BOOT_STRUCT boot;		
32		/*		
33		* Required variable to allow the Lance diagnostics to use sendto()		
34		* with small (runt) packets.		
35		*/		
36		u_short allow_runt_packets = FALSE;		
37		/*		
38		* Structure definition and allocation of permanent memory		
39		* for the desired number of receive buffers.		
40		*/		
41		typedef struct {		
42		char buf[MAX_RECEIVE_BUFFER_SIZE];		
43		} RECV_BUF;		
44		static RECV_BUF lm_receive_buffers(MAX_RECEIVE_BUFFERS + MAX_USERS);		
45		static RECV_BUF *p_lm_rcv_buffers_extra( MAX_USERS );		
46		static RECV_MDE extra_rcv_desc_buffers( MAX_USERS );		
47		static unsigned char extra_rcv_desc_count = 0;		
48		static unsigned char in_extra_rcv_desc_ptr = 0;		
49		static unsigned char out_extra_rcv_desc_ptr = 0;		
50		static char too_many_buffers = 0;		
51		static char available_extra_buffer = (MAX_USERS - 1);		
52		/*		
53		* Note that the following character array space is 8 bytes larger than		
54		* what is really required in order to get space that is aligned on a		
55		* quadword (8 byte) boundary. The stupid SUN assembler has no means for		
56		* allowing me to specify this directly.		
57		*/		
58		/*		
59		* Also note that each entry mapped into the table must start on a		
60		* quadword (8 byte) boundary. This automatically happens because of		
61		* the sizeof("MDE") being equal to 8 bytes.		
62		*/		
63		static char rcv_desc_ring_memory[8 + MAX_RECEIVE_BUFFERS * sizeof(RECV_MDE)];		
64		static char trans_desc_ring_memory[8 + MAX_TRANSMIT_BUFFERS * sizeof(TRANS_MDE)];		
65		/*		
66		* Macro for returning a pointer value that is aligned on a quad-word boundary.		
67		*/		
68		#define QUAD_WORD_ALIGN(X) ((8 * (((u_long) (X) + 8) / 8))		
69		/*		
70		* Pointers to keep track of our notion of the beginning, current and end		
71		* of the receive and transmit descriptor message rings. These pointers		
72		* are given values when the receive and transmit rings are initialized.		
73		*/		
74		static RECV_MDE *minimum_receive_pointer;		
75		static RECV_MDE *current_receive_pointer;		
76		static RECV_MDE *maximum_receive_pointer;		
77		static TRANS_MDE *minimum_transmit_pointer;		
78		static TRANS_MDE *current_transmit_pointer;		
79		static TRANS_MDE *maximum_transmit_pointer;		
80		static TRANS_MDE *lance_send_packet;		
81		extern u_short initialize_lance();		
82		extern u_short set_up_control_and_status_registers();		
83		extern void set_up_extra_buffers();		
84		extern void memory_align_ring_descriptors();		
85		extern void initialize_transmit_buffers();		
86		extern void initialize_receive_buffers();		
87		extern void output_routine();		
88		extern void set_up_initialization_block();		
89		static void memory_copy();		
90		extern void lance_isr_receive();		
91		extern u_short lance_receive();		
92		extern u_short lance_transmit();		
93		extern u_short process_transmit_interrupt();		
94		extern u_short process_receive_interrupt();		
95		int lm_intr_requested; /* global interrupt advisory flag */		
96		/*		
97		* Properly reset the LANCE using the CPU control register bit.		
98		* Leaves it reset, or may be followed by turning off the lance reset		
99		* to provide a reset_lance() function.		
100		*/		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 2/18
LINE #	SOURCE TEXT			
121	void			
122	shutdown_lance()			
123	{			
124	register cpu_control_reg_struct *control_reg;			
125	register u_long delay_count;			
126	}			
127	/*			
128	The LANCE may very well be doing something. In that event, we			
129	don't want to risk confusing its requester and possibly the CPU			
130	arbitrar, so we let it complete. In order to guarantee that it			
131	doesn't start any master cycles, turn off the lance enable bit.			
132	Then delay a generous amount of time to allow for a worst-case			
133	burst. These seminally require 6-bus, so we'll allow 50us. After			
134	all, we're not in THAT much of a hurry here. Then pound on the			
135	reset bit and hold it. Though the Am7990 specs just 2 clock cycles			
136	minimum low time (i.e. 200ns), we drive it for a more comfortable			
137	5us. No real good reason.			
138	*/			
139	control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
140	control_reg->lance_enable = 0;			
141	delay_microseconds(delay_count, 50);			
142	control_reg->not_lance_reset = 0;			
143	delay_microseconds(delay_count, 5);			
144	}			
145	/*			
146	That packets are used only in diagnosis for loopback tests.			
147	We shut off this flag here just in case it got set spuriously.			
148	*/			
149	allow_rmt_packets = FALSE;			
150	}			
151	/*			
152	Shut down the LANCE, then allow it to come out of reset.			
153	*/			
154	void			
155	shutdown_and_reset_lance()			
156	{			
157	register cpu_control_reg_struct *control_reg;			
158	control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;			
159	shutdown_lance();			
160	control_reg->not_lance_reset = 1;			
161	}			
162	/*			
163	Set up the LANCE to receive and transmit data.			
164	*/			
165	void			
166	set_lance_ready_to_go()			
167	{			
168	RECVR_NODE       *rcvr_ring;			
169	TRANSM_NODE     *trans_ring;			
170	int intr_requested = 0;			
171	shutdown_and_reset_lance();			
172	set_up_extra_buffers();			
173	memory_align_ring_descriptors(&rcvr_ring, &trans_ring);			
174	initialize_transmit_buffers(trans_ring);			
175	initialize_receive_buffers(rcvr_ring);			
176	if (initialize_lance(rcvr_ring, trans_ring) == FAILURE)			
177	return(FAILURE);			
178	return(SUCCESS);			
179	}			
180	/*			
181	Set up extra buffers.			
182	*/			
183	void			
184	set_up_extra_buffers()			
185	{			
186	register i;			
187	for( i=0; i < MAX_USERS; i++ ) {			
188	p_lm_rcv_buffers_extra[ i ] = &lm_receive_buffers[ i + MAX_RECEIVE_BUFFERS ];			
189	}			
190	}			
191	/*			
192	Start the LANCE.			
193	*/			
194	void			
195	start_lance()			
196	{			
197	u_short value;			
198	register       u_short i;			
199	}			
200	/*			
201	Tell the LANCE to initialize itself.			
202	*/			
203	if (write_lance_csr(SELECT_CSRO, INITIALIZE_START) == FAILURE)			
204	return(FAILURE);			
205	/*			
206	We must wait for the LANCE to finish its			
207	initialization before we continue processing.			
208	*/			
209	for( i=0; i<MAX_INITIALIZED_DONE_POLLS; ++i ) {			
210	lm_delay(3);			
211	if (read_lance_csr(SELECT_CSRO, &value) == FAILURE)			
212	return(FAILURE);			
213	if ((value & INITIALIZE_DONE) == INITIALIZE_DONE)			
214	break;			
215	}			
216	/*			
217	If the LANCE still hasn't initialized itself, give up.			
218	*/			
219	if ((value & INITIALIZE_DONE) != INITIALIZE_DONE)			
220	return(FAILURE);			
221	/*			
222	Let's get the show on the road.			
223	*/			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89 TIME 4:42:12 pm	PAGE # 3/19
LINE #	SOURCE TEXT			
241	if (write_lance_car(SELECT_CS0, START_ACTIVITY) == FAILURE)			
242	return(FAILURE);			
243	/*			
244	* Things appear to work better (at all?) if we wait a little bit here.			
245	*/			
246	lm_delay(10);			
247	/*			
248	* Enable LANCE interrupts.			
249	*/			
250	if (write_lance_car(SELECT_CS0, INTERRUPT_ENABLE) == FAILURE)			
251	return(FAILURE);			
252	/*			
253	* Things appear to work better (at all?) if we wait a little bit here.			
254	*/			
255	lm_delay(10);			
256	return(SUCCESS);			
257	}			
258	/*			
259	* The following function is used by the CPU diagnostics to have the			
260	* LANCE read the initialization block but not start activity. It must			
261	* not use any VMT calls, e.g. lm_delay().			
262	*/			
263	u_short			
264	start_lance_initialize_only()			
265	{			
266	u_short i;			
267	u_short value;			
268	register u_long bogus;			
269	/*			
270	* Tell the LANCE to initialize itself.			
271	*/			
272	if (write_lance_car(SELECT_CS0, (u_short)INITIALIZE_START) == FAILURE)			
273	return (FAILURE);			
274	/*			
275	* We must wait for the LANCE to finish its initialization before we			
276	* continue processing.			
277	*/			
278	for (i = 0; i < MAX_INITIALIZE_DONE_POLLS; ++i)			
279	{			
280	delay_milliseconds(bogus, 3);			
281	if (read_lance_car(SELECT_CS0, &value) == FAILURE)			
282	return (FAILURE);			
283	if ((value & INITIALIZE_DONE) == INITIALIZE_DONE)			
284	break;			
285	}			
286	/*			
287	* If the LANCE still hasn't initialized itself, give up.			
288	*/			
289	if ((value & INITIALIZE_DONE) != INITIALIZE_DONE)			
290	return (FAILURE);			
291	return(SUCCESS);			
292	}			
293	/*			
294	* Quad-word align the transmit and receive message descriptor rings.			
295	* This is a LANCE requirement that the SUN assembler can not provide.			
296	*/			
297	if ((u_long) trans_desc_ring_memory & 8 == 0)			
298	trans = (TRANS_NDE *) trans_desc_ring_memory;			
299	else trans = (TRANS_NDE *) QUAD_WORD_ALIGN(trans_desc_ring_memory);			
300	if ((u_long) recv_desc_ring_memory & 8 == 0)			
301	recv = (RCV_NDE *) recv_desc_ring_memory;			
302	else recv = (RCV_NDE *) QUAD_WORD_ALIGN(recv_desc_ring_memory);			
303	/*			
304	* Initialize transmit buffers (trans)			
305	*/			
306	register TRANS_NDE *trans;			
307	{			
308	register u_short i;			
309	/*			
310	* Set an unchanging pointer to the beginning of the transmit			
311	* message descriptor ring so that we know where to wrap around			
312	* to as more and more messages are sent (see lance_transmit()).			
313	*/			
314	minimize_transmit_pointer = trans;			
315	/*			
316	* Set curr, as opposed to the LANCE's, notion of what the current			
317	* transmit message descriptor is.			
318	*/			
319	current_transmit_pointer = trans;			
320	lance_host_packet = trans;			
321	for(i=0; i<MAX_TRANSMIT_BUFFERS; ++i) {			
322	/* No other fields in the transmit message descriptor */			
323	/* need be initialized at this time. They are all set */			
324	/* when the to be transmitted packet buffer is set up. */			
325	trans->must_be_ones = 0xFF;			
326	trans->own_buffer = HOST_BUFFER_OWNERSHIP;			
327	++trans; /* advance to the next transmit descriptor. */			
328	}			
329	/*			
330	* Set an unchanging pointer to the end of the transmit			
331	* message descriptor ring so that we know when to wrap around			
332	* as more and more messages are sent (see lance_transmit()).			
333	*/			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 4/20
LINE #		SOURCE TEXT		
361		maximum_transmit_pointer = trans;		
362		}		
363		}		
364		void		
365		initialize_receive_buffers(recv)		
366		register RECV_MODE *recv;		
367		{		
368		register u_short i;		
369		{		
370		/*		
371		* Set an unchanging pointer to the beginning of the receive		
372		* message descriptor ring so that we know where to wrap around		
373		* to as more and more messages are received (see lance_receive()).		
374		*/		
375		minimum_receive_pointer = recv;		
376		/*		
377		* Set our, as opposed to the LANCE's, notion of what the current		
378		* receive message descriptor is.		
379		*/		
380		current_receive_pointer = recv;		
381		for(i=0; i<MAX_RECEIVE_BUFFERS; ++i) {		
382		/* Set receive message descriptor 0 */		
383		recv->low_buffer_address = (u_short) &lance_receive_buffers[i];		
384		/* Set receive message descriptor 1 */		
385		recv->own_buffer = LANCE_BUFFER_OWNERSHIP;		
386		recv->error_summary = 0;		
387		recv->framing_error = 0;		
388		recv->overflow_error = 0;		
389		recv->crc_error = 0;		
390		recv->buffer_error = 0;		
391		recv->start_of_packet = 0;		
392		recv->end_of_packet = 0;		
393		recv->high_buffer_address = (unsigned) &lance_receive_buffers[i] >> 16;		
394		/* Set receive message descriptor 2 */		
395		recv->must_be_ones = 0x1;		
396		recv->buffer_byte_count = MAX_RECEIVE_BUFFER_SIZE;		
397		/* Set receive message descriptor 3 */		
398		recv->must_be_zeros = 0;		
399		recv->message_byte_count = 0;		
400		++recv; /* advance to the next receive descriptor. */		
401		/*		
402		* Set an unchanging pointer to the end of the receive		
403		* message descriptor ring so that we know when to wrap around		
404		* as more and more messages are received (see lance_receive()).		
405		*/		
406		maximum_receive_pointer = recv;		
407		}		
408		}		
409		u_short		
410		initialize_lance(recv, trans)		
411		register RECV_MODE *recv;		
412		register TRANS_MODE *trans;		
413		{		
414		if (set_up_control_and_status_registers() == FAILURE)		
415		return(FAILURE);		
416		set_up_initialization_block(recv, trans);		
417		return(SUCCESS);		
418		}		
419		}		
420		static u_short		
421		set_up_control_and_status_registers()		
422		{		
423		/*		
424		* The LANCE requires that the STOP bit of CS0 be set		
425		* before accessing CS0, CS1, or CS2.		
426		*/		
427		if (write_lance_car(SELECT_CS0, STOP_ACTIVITY) == FAILURE)		
428		return(FAILURE);		
429		/*		
430		* The CPU requires that byte control and swap be set.		
431		*/		
432		if (write_lance_car(SELECT_CS3, BYTE_CONTROL BYTE_SWAP) == FAILURE)		
433		return(FAILURE);		
434		/*		
435		* Load in the address of the initialization block.		
436		*/		
437		if (write_lance_car(SELECT_CS2, (u_short) (0xFF & ((u_long) init_block_mem >> 16))) == FAILURE)		
438		return(FAILURE);		
439		if (write_lance_car(SELECT_CS1, (u_short) (0xFF & ((u_long) init_block_mem >> 16))) == FAILURE)		
440		return(FAILURE);		
441		return(SUCCESS);		
442		}		
443		static void		
444		set_up_initialization_block(recv, trans)		
445		register RECV_MODE *recv;		
446		register TRANS_MODE *trans;		
447		{		
448		register u_char *set_address;		
449		register INIT_BLOCK *init_block;		
450		extern ID_PROM_CPU id_prom;		
451		{		
452		init_block = (INIT_BLOCK *) init_block_mem;		
453		/* Set up the correct operational modes */		
454		init_block->promiscuous = 0;		
455		init_block->reserved_1 = 0;		
456		init_block->internal_loopback = 0;		
457		init_block->disable_retry = 0;		
458		}		

1775

5,353,243

1776

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/lance.c

DATE 5/23/89 PAGE #  
TIME 4:42:12 pm 5/21

```

LINE # SOURCE TEXT
481 init_block->force_collision = 0;
482 init_block->disable_crc_transmit = 0;
483 init_block->loopback = 0;
484 init_block->disable_transmit = 0;
485 init_block->disable_receive = 0;
486
487 /* Set up the 48-bits for the physical Ethernet address */
488 /* Note the byte sequence (see lance.h for details) */
489 const_address = (u_char *) id_prom.ethernet;
490 init_block->physical_address_1 = *const_address++;
491 init_block->physical_address_2 = *const_address++;
492 init_block->physical_address_3 = *const_address++;
493 init_block->physical_address_4 = *const_address++;
494 init_block->physical_address_5 = *const_address++;
495 init_block->physical_address_6 = *const_address++;
496
497 /* Set up the logical address filter */
498 /* Must always be zero (0) as we don't implement multi-casting */
499 init_block->logical_address_1 = 0;
500 init_block->logical_address_2 = 0;
501
502 /* Set up the receive descriptor ring pointer */
503 init_block->low_receive_ring_address = (u_short) recv;
504 init_block->high_receive_ring_address = (unsigned) recv >> 16;
505 init_block->receive_ring_length = POWER_OF_2_MAX_RECEIVE_BUFFERS;
506 init_block->reserved_3 = 0;
507
508 /* Set up the transmit descriptor ring pointer */
509 init_block->low_transmit_ring_address = (u_short) trans;
510 init_block->high_transmit_ring_address = (unsigned) trans >> 16;
511 init_block->transmit_ring_length = POWER_OF_2_MAX_TRANSMIT_BUFFERS;
512 init_block->reserved_3 = 0;
513
514
515 /*
516 ** This is the fast access method to I/O CSRs
517 ** The interrupt routine calls this to speed up access to the lance
518 ** It reads CSR0 then clears out bits and enables INTs
519 ** From Lance.
520 */
521 u_short
522 isr_read_write_lance_car( value )
523 register u_short *value;
524 {
525     register u_short i;
526     register u_short *lance_data_register;
527     register u_short *lance_address_register;
528
529     register cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG;
530
531     /*
532     * Initialize our local register variables.
533     */
534     lance_data_register = (u_short *) CPU_LANCE_DATA_REG;
535     lance_address_register = (u_short *) CPU_LANCE_ADDR_REG;
536
537     /*
538     * Don't change this code unless you carefully examine the assembler
539     * that is produced to ensure that the code will properly handle
540     * the hardware idiosyncrasies of the LANCE. Talk with Mark if
541     * you have any questions regarding the LANCE's functionality.
542     */
543
544     i = MAX_LANCE_ENABLE_POLL;
545     cpu_control_register->lance_enable = 0;
546     while (cpu_control_register->lance_busy == 1) {
547         if (--i) {
548             cpu_control_register->lance_enable = 1; /* Why? */
549             return(FAILURE);
550         }
551     }
552
553     *lance_address_register = SELECT_CSR0;
554     *value = *lance_data_register;
555
556     *lance_data_register = *value | INTERRUPT_ENABLE;
557     cpu_control_register->lance_enable = 1;
558
559     return(SUCCESS);
560 }
561
562
563 u_short
564 write_lance_car(which_car, value)
565 register u_short which_car;
566 register u_short value;
567 {
568     register u_short i;
569     register u_short *lance_data_register;
570     register u_short *lance_address_register;
571     register u_short interrupts_were_enabled;
572
573     register cpu_control_reg_struct *cpu_control_register = (cpu_control_reg_struct *) CPU_CONTROL_REG;
574
575     /*
576     * Initialize our local register variables.
577     */
578     lance_data_register = (u_short *) CPU_LANCE_DATA_REG;
579     lance_address_register = (u_short *) CPU_LANCE_ADDR_REG;
580
581     /*
582     * Don't change this code unless you carefully examine the assembler
583     * that is produced to ensure that the code will properly handle
584     * the hardware idiosyncrasies of the LANCE. Talk with Mark if
585     * you have any questions regarding the LANCE's functionality.
586     */
587
588     /*
589     * MUST disable interrupts during this time as the data that
590     * gets written depends upon which data port is selected by
591     * the CPU_LANCE_ADDR_REG, among other things.
592     */
593     interrupts_were_enabled = cpu_control_register->global_intr_ena;
594     CPU_DISABLE_INTERRUPTS;
595
596     /*
597     * Inhibit the LANCE from doing anything while we get at its CSR.
598     */
599
600

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 6/22
LINE #	SOURCE TEXT			
601	/* Then test several times for the LANCE being busy. When its			
602	/* free, look at the value in the LANCE's data register.			
603	/* The address port must be written before the data port.			
604	/* Note that we reset the LANCE's address port to reference			
605	/* CSRS. See the comment above for this "feature's" motivation.			
606	/*			
607	/* Enable the LANCE after we're done accessing it's CSR.			
608	/*			
609	/*			
610	i = MAX_LANCE_ENABLE_POLLS;			
611	cpu_control_register->lance_enable = 0;			
612	while (cpu_control_register->lance_busy == 1) {			
613	if (!--i) {			
614	cpu_control_register->lance_enable = 1; /* Why? */			
615	return(FAILURE);			
616	}			
617	}			
618	/*			
619	/*lance_data_register = which_car;			
620	/*lance_data_register = value;			
621	cpu_control_register->lance_enable = 1;			
622	/*			
623	/* Turn interrupts back on, only if they were enabled before.			
624	/*			
625	/*			
626	if (interrupts_were_enabled)			
627	CPU_ENABLE_INTERRUPTS;			
628	/*			
629	return(SUCCESS);			
630	}			
631	/*			
632	/*			
633	u_short			
634	read_lance_car(which_car, value)			
635	register u_short which_car;			
636	register u_short *value;			
637	{			
638	register u_short i;			
639	register u_short *lance_data_register;			
640	register u_short *lance_address_register;			
641	register u_short interrupts_were_enabled;			
642	/*			
643	register cpu_control_reg_struct *cpu_control_register= (cpu_control_reg_struct *) CPU_CONTROL_REG;			
644	/*			
645	/*			
646	/*			
647	/* Initialize our local register variables.			
648	/*			
649	lance_data_register = (u_short *) CPU_LANCE_DATA_REG;			
650	lance_address_register = (u_short *) CPU_LANCE_ADDR_REG;			
651	/*			
652	/*			
653	/* Don't change this code unless you carefully examine the assembler			
654	/* that is produced to ensure that the code will properly handle			
655	/* the hardware idiosyncrasies of the LANCE. Talk with Mark if			
656	/* you have any questions regarding the LANCE's functionality.			
657	/*			
658	/*			
659	/*			
660	/* MUST disable interrupts during this time as the data that			
661	/* gets written depends upon which data port is selected by			
662	/* the CPU_LANCE_ADDR_REG, among other things.			
663	/*			
664	interrupts_were_enabled = cpu_control_register->global_intr_ena;			
665	CPU_DISABLE_INTERRUPTS;			
666	/*			
667	/*			
668	/* Inhibit the LANCE from doing anything while we get at its CSR.			
669	/*			
670	/*			
671	/* Then test several times for the LANCE being busy. When its			
672	/* free, look at the value in the LANCE's data register.			
673	/* The address port must be written before the data port.			
674	/* Note that we reset the LANCE's address port to reference			
675	/* CSRS. See the comment above for this "feature's" motivation.			
676	/*			
677	/* Enable the LANCE after we're done accessing it's CSR.			
678	/*			
679	/*			
680	i = MAX_LANCE_ENABLE_POLLS;			
681	cpu_control_register->lance_enable = 0;			
682	while (cpu_control_register->lance_busy == 1) {			
683	if (!--i) {			
684	cpu_control_register->lance_enable = 1; /* Why? */			
685	return(FAILURE);			
686	}			
687	}			
688	/*			
689	/*lance_data_register = which_car;			
690	/*value = *lance_data_register;			
691	cpu_control_register->lance_enable = 1;			
692	/*			
693	/* Turn interrupts back on, only if they were enabled before.			
694	/*			
695	/*			
696	if (interrupts_were_enabled)			
697	CPU_ENABLE_INTERRUPTS;			
698	/*			
699	return(SUCCESS);			
700	/*			
701	/*			
702	int			
703	sendto(id, message, length, flag, sock, sock_size)			
704	int id, /* ignored */			
705	register char *message;			
706	int length;			
707	int flag;			
708	register SOCKET_ADDRESS *sock;			
709	int sock_size, /* ignored */			
710	{			
711	u_long error;			
712	/*			
713	register ETHERNET_HEADER *enet_header;			
714	register IP_HEADER *ip_header;			
715	register UDP_HEADER *udp_header;			
716	register INIT_BLOCK *init_block;			
717	/*			
718	extern u_short compute_ip_checksum();			
719	/*			
720	/*			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/lance.c

DATE 5/23/89 PAGE #  
TIME 4:42:12 pm 7/23

```

LINE #
721 /* Initialize the network headers on the outgoing message.
722
723 enet_header = (ETHERNET_HEADER *) message;
724
725
726 /* Copy the Ethernet source address from the LANCE's
727 * initialization block to the outgoing message buffer.
728
729 init_block = (INIT_BLOCK *) init_block_ptr;
730 enet_header->source[0] = init_block->physical_address_1;
731 enet_header->source[1] = init_block->physical_address_2;
732 enet_header->source[2] = init_block->physical_address_3;
733 enet_header->source[3] = init_block->physical_address_4;
734 enet_header->source[4] = init_block->physical_address_5;
735 enet_header->source[5] = init_block->physical_address_6;
736
737
738 /* Copy the Ethernet destination address from the
739 * message socket to the outgoing message buffer.
740
741 enet_header->destination[0] = sock->destination[0];
742 enet_header->destination[1] = sock->destination[1];
743 enet_header->destination[2] = sock->destination[2];
744 enet_header->destination[3] = sock->destination[3];
745 enet_header->destination[4] = sock->destination[4];
746 enet_header->destination[5] = sock->destination[5];
747
748
749 /* Copy the packet type from the message socket to the
750 * outgoing message buffer.
751
752 enet_header->type = sock->packet_type;
753
754 if ((flag & UDP_IP_PACKET) == UDP_IP_PACKET) {
755     ip_header = (IP_HEADER *) (message + sizeof(ETHERNET_HEADER));
756     udp_header = (UDP_HEADER *) (message + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER));
757
758     ip_header->version_and_header_length = 0x45; /* IP */
759     ip_header->type_of_service = 0x00; /* not used */
760     ip_header->total_length = length + sizeof(IP_HEADER) + sizeof(UDP_HEADER); /* length of packet w/ headers */
761     ip_header->identification = 0x0000; /* not used */
762     ip_header->fragment_offset = 0x0000; /* not used */
763     ip_header->time_to_live = 0x1f; /* 255 seconds */
764     ip_header->protocol = 0x11; /* UDP */
765     ip_header->checksum = 0x0000; /* computed later */
766     if (((flag & ALIVE_PACKET) == ALIVE_PACKET) || ((flag & DEATH_PACKET) == DEATH_PACKET)) {
767         ip_header->source_address = alive_ipnet;
768     } else {
769         ip_header->source_address = modeler_ipnet;
770     }
771     ip_header->destination_address = sock->address;
772     ip_header->checksum = compute_ip_checksum((u_short *) ip_header);
773     if ((flag & ALIVE_PACKET) == ALIVE_PACKET) {
774         udp_header->source_port = MODELERS_ARE_YOU_ALIVE_PORT;
775     } else if ((flag & DEATH_PACKET) == DEATH_PACKET) {
776         udp_header->source_port = MODELERS_DEATH_PORT;
777     } else {
778         udp_header->source_port = MODELERS_ETHERNET_PORT;
779     }
780     udp_header->destination_port = sock->port;
781     udp_header->length = (u_short) (length + sizeof(UDP_HEADER));
782     udp_header->checksum = 0x0000; /* Not used */
783     if (lance_transmit(message, (u_short) (length + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER) + sizeof(UDP_HEADER)), &error) != SUCCESS) {
784         return(-error); /* Must return negative number to indicate error. */
785     }
786 } else { /* RAW PACKET */
787     if (lance_transmit(message, (u_short) (length + sizeof(ETHERNET_HEADER)), &error) != SUCCESS) {
788         return(-error); /* Must return negative number to indicate error. */
789     }
790 }
791
792 return(length);
793
794
795
796 static u_short
797 compute_ip_checksum (a)
798 register u_short *a;
799 {
800     /*
801     * This function computes the IP header checksum. The header is passed in as
802     * an array of unsigned shorts. This function should be called only after all
803     * of the bytes have been appropriately swapped. The resulting checksum must
804     * NOT be byte swapped. (Stolen from KF's VMS network code.)
805     */
806     register u_long sum;
807
808     /* Sum up the 10 words (20 bytes) of the ip header. */
809     sum = *a;
810     sum += **a;
811     sum += **a;
812     sum += **a;
813     sum += **a;
814     sum += **a;
815     sum += **a;
816     sum += **a;
817     sum += **a;
818     sum += **a;
819     sum += **a;
820     sum += **a;
821     sum += sum >> 16; /* add in the carry */
822     return((u_short) ~sum); /* returns the one's complement of the checksum */
823
824
825
826
827 int
828 recvfrom (fd, message, length, option, from, from_len)
829 int fd;
830 char *message;
831 int length;
832 int option;
833 SOCKET_ADDRESS *from;
834 int *from_len;
835 {
836     u_long error;
837     u_short error_code, actual_length;
838     int err;
839 }

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 8/24
LINE #	SOURCE TEXT			
840	register       ETHERNET_HEADER *enet_header;			
841	register       RCV_MDE *rcv;			
842	register       IP_HEADER *ip_header;			
843	register       UDP_HEADER *udp_header;			
844	register       struct ethernet_arp_packet *packet;			
845				
846				
847	while( 1 )			
848	{			
849	/*			
850	** wait for a message from host			
851	*/			
852	CPU_DISABLE_INTERRUPTS;			
853	while( !extra_rcv_desc_count )			
854	{			
855	CPU_ENABLE_INTERRUPTS;			
856	if( option == RECVRON_TIMEOUT )			
857	ac_spnd( rcv_lance_pkt_semaphore, SECOND * TIMEOUT, &err );			
858	else			
859	ac_spnd( rcv_lance_pkt_semaphore, 0, &err );			
860	if( err )			
861	return( -1 );			
862	}			
863	CPU_ENABLE_INTERRUPTS;			
864	/*			
865	** there is something from a host			
866	*/			
867	rcv = extra_rcv_desc_buffers[ out_extra_rcv_desc_ptr ];			
868				
869	/*			
870	** get message			
871	*/			
872	if ( lance_receive(message, (u_short)length,			
873	&actual_length, rcv, option, &error) != SUCCESS)			
874	{			
875	if( option != MSG_PEEK )			
876	{			
877	CPU_DISABLE_INTERRUPTS;			
878	extra_rcv_desc_count--;			
879	out_extra_rcv_desc_ptr++;			
880	out_extra_rcv_desc_ptr %= ( MAX_USERS - 1 );			
881	p_lm_rcv_buffers_extra[ ++available_extra_buffer ]			
882	= (RCV_BUF *)((rcv->high_buffer_address << 16)			
883	rcv->low_buffer_address);			
884	if( too_many_buffers == 1)			
885	lance_isr_receive( &error_code );			
886	CPU_ENABLE_INTERRUPTS;			
887	}			
888	/* Next returns negative number to indicate error. */			
889	if (error <= 0)			
890	return(-1);			
891	else return(-error);			
892	}			
893	if( option != MSG_PEEK )			
894	{			
895	CPU_DISABLE_INTERRUPTS;			
896	extra_rcv_desc_count--;			
897	out_extra_rcv_desc_ptr++;			
898	out_extra_rcv_desc_ptr %= ( MAX_USERS - 1 );			
899	p_lm_rcv_buffers_extra[ ++available_extra_buffer ]			
900	= (RCV_BUF *)((rcv->high_buffer_address << 16)			
901	rcv->low_buffer_address);			
902	if( too_many_buffers == 1)			
903	lance_isr_receive( &error_code );			
904	CPU_ENABLE_INTERRUPTS;			
905	}			
906				
907	udp_header = (UDP_HEADER *) (message+sizeof(ETHERNET_HEADER)+sizeof(IP_HEADER));			
908	ip_header = (IP_HEADER *) (message + sizeof(ETHERNET_HEADER));			
909				
910	packet = (struct ethernet_arp_packet *) message;			
911	if ( packet->arp.arp_opcode == ARP_REPLY )			
912	{			
913	return(sizeof(struct ethernet_arp_packet));			
914	}			
915	enet_header = (ETHERNET_HEADER *) message;			
916	from->packet_type = enet_header->type;			
917	from->destination[0] = enet_header->source[0];			
918	from->destination[1] = enet_header->source[1];			
919	from->destination[2] = enet_header->source[2];			
920	from->destination[3] = enet_header->source[3];			
921	from->destination[4] = enet_header->source[4];			
922	from->destination[5] = enet_header->source[5];			
923				
924	from->port = udp_header->source_port;			
925	from->address = ip_header->source_address;			
926	from->family = AF_INET;			
927				
928	modeler_inet_address = ip_header->destination_address;			
929				
930	from_len = SOCK_SIZE;			
931				
932	actual_length = ip_header->total_length - (sizeof( IP_HEADER ) + sizeof( UDP_HEADER )); /* length of packet w/o headers */			
933				
934	/*			
935	** if we get a packet with funny insides just continue.			
936	*/			
937	if( actual_length > MAX_RECEIVE_BUFFER_SIZE    actual_length == 0)			
938	continue;			
939				
940	return(actual_length);			
941	}			
942				
943				
944				
945	u_short			
946	lance_transmit(message, length, error)			
947	register       char *message;			
948	register       u_short length;			
949	register       u_long *error;			
950	{			
951	register       TRANS_MDE *trans;			
952	register       u_short interrupts_were_enabled;			
953				
954				
955	/*			
956	* Start out with no errors.			
957	*/			
958	*error = 0;			
959				

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89 TIME 4:42:12 pm	PAGE # 9/25
LINE #	SOURCE TEXT			
960	interrupts_were_enabled =			
961	((cps_control_reg_struct *) CPU_CONTROL_REG->global_intr_ess,			
962	CPU_DISABLE_INTERRUPTS,			
963				
964	/* Local register pointer.			
965	*/			
966	trans = current_transmit_pointer;			
967				
968	/*			
969	* Set curr. as opposed to the LANCE's, notion of what the current transmit			
970	* message descriptor is. Actually, this is the next transmit descriptor			
971	* to be used, but I think the code is clearer if it's name is "current".			
972	*/			
973	/*			
974	* The transmit message descriptors are in a ring structure which causes			
975	* us to have to wrap around to the beginning when the end of the list			
976	* is reached.			
977	*/			
978	if (++current_transmit_pointer >= maximum_transmit_pointer) {			
979	current_transmit_pointer = minimum_transmit_pointer;			
980	}			
981				
982	if (interrupts_were_enabled)			
983	CPU_ENABLE_INTERRUPTS,			
984				
985	if( trans->own_buffer == LANCE_BUFFER_OWNERSHIP )			
986	return( FAILURE );			
987				
988	/*			
989	* Set up the address of the message that is being sent, indicate that			
990	* this is both the start and the end of this packet, set the packet			
991	* byte count, finally - note that this must be done last - the			
992	* buffer is turned over to the LANCE for transmission.			
993	*/			
994	trans->low_buffer_address = (u_short) message;			
995	trans->high_buffer_address = (unsigned) message >> 16;			
996	trans->start_of_packet = 1;			
997	trans->end_of_packet = 1;			
998				
999	/*			
1000	** most hosts will not like very short packets			
1001	** 64 is a minimum for SUN's			
1002	*/			
1003	if (allow_runt_packets == FALSE) {			
1004	if (length < 64) {			
1005	length = 64;			
1006	}			
1007				
1008				
1009	trans->buffer_byte_count = (unsigned) (1 + "length);			
1010	trans->own_buffer = LANCE_BUFFER_OWNERSHIP;			
1011				
1012	/*			
1013	* Poke the LANCE to let it know that we have a message for it to send.			
1014	* Otherwise, it would wait until the polling time interval had elapsed.			
1015	* Note that we must set the interrupt enable bit ON every time the			
1016	* transmit demand bit is set, because for some unknown reason, interrupts			
1017	* are disabled whenever we set transmit demand.			
1018	*/			
1019	if (write_lance_csr(SELECT_CSR0, TRANSMIT_DEMAND INTERUPT_ENABLE) == FAILURE) {			
1020	error  = FAILURE_TO_WRITE_TO_LANCE_CSR;			
1021	return(FAILURE);			
1022				
1023				
1024	return(SUCCESS);			
1025				
1026				
1027	static u_short			
1028	lance_receive(buffer, max_length, actual_length, rcv, options, error)			
1029	register char *buffer;			
1030	register u_short max_length;			
1031	register u_short *actual_length;			
1032	register u_long *error;			
1033	register int options;			
1034	register RECV_MSG *rcv;			
1035	{			
1036	register char *rcv_buffer;			
1037	/*			
1038	* Start out with no errors.			
1039	*/			
1040	*error = 0;			
1041				
1042				
1043	/*			
1044	* This implementation of the LANCE code requires that the entire			
1045	* message fit within a single receive message buffer. Check that			
1046	* the LANCE thinks that this has happened.			
1047	*/			
1048	if (rcv->start_of_packet != 1)			
1049	*error  = PACKET_OVERFLOW_ERROR;			
1050	if (rcv->end_of_packet != 1)			
1051	*error  = PACKET_OVERFLOW_ERROR;			
1052				
1053	if (*error != 0)			
1054	return(FAILURE);			
1055				
1056	/*			
1057	* Looks like we have a good message. Let's process and return it			
1058	* to the anxious user.			
1059	*/			
1060				
1061	/*			
1062	* Check to see if the size of the received message is small enough			
1063	* to fit into the buffer that the user passed us. The number of			
1064	* bytes that the LANCE deals with is 4 more than one would expect			
1065	* because of the 4 byte CRC. Adjust the actual length as indicated.			
1066	*/			
1067	*actual_length = rcv->message_byte_count-4;			
1068	if( options != MSG_PEEK )			
1069	{			
1070	if (*actual_length > max_length)			
1071	{			
1072	*error  = BUFFER_TOO_SMALL;			
1073	return(FAILURE);			
1074	}			
1075				
1076				
1077	/*			
1078	* Copy the received buffer to the user's buffer.			
1079	*/			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 10/26
LINE #	SOURCE TEXT			
1080	recv_buffer = (char *) ((recv->high_buffer_address << 16)   recv->low_buffer_address);			
1081	/*			
1082	* If the packet is not a broadcast, copy the data from the LANCE's buffer			
1083	* to the calling routine's buffer. Broadcast packets are indicated by			
1084	* a returned length of 0 - no data is copied.			
1085	*/			
1086	if( option != MSG_PEEK )			
1087	memory_copy((u_long *)buffer, (u_long *)recv_buffer, (short)*actual_length);			
1088	else			
1089	memory_copy((u_long *)buffer, (u_long *)recv_buffer, (short)max_length);			
1090	/*			
1091	* We must adjust the length of the actual message to compensate for the			
1092	* calling routines lack of knowledge of the Ethernet header.			
1093	*/			
1094	*actual_length -- sizeof(ETHERNET_HEADER);			
1095				
1096	return(SUCCESS);			
1097				
1098	}			
1099				
1100				
1101	/*			
1102	* The code has been changed so that it is not possible for lance_isr_receive()			
1103	* to return any failures. If the calling routine wants to see if "warnings"			
1104	* occurred, they can look at the returned "error".			
1105	*/			
1106	static void			
1107	lance_isr_receive( error )			
1108	register u_short *error;			
1109	{			
1110	register char *recv_buffer;			
1111	register			
1112	RCV_HDR *rcv;			
1113	register IP_HEADER *ip_header;			
1114	register UDP_HEADER *udp_header;			
1115	register LM_HEADER *lm_header;			
1116	register ICMP_HEADER *icmp_pkt;			
1117				
1118	extern char lm_valid_address;			
1119	extern void process_other_stuff();			
1120	extern void process_are_you_alive();			
1121				
1122	/*			
1123	* Start out with no errors.			
1124	*/			
1125	*error = 0;			
1126				
1127	/*			
1128	* Note: in the following code we must wait until the last possible			
1129	* moment to turn the buffer back over to the LANCE. Therefore, don't			
1130	* execute the following line of code until you have to:			
1131	*/			
1132	/* recv->own_buffer = LANCE_BUFFER_OWNERSHIP; */			
1133				
1134				
1135				
1136	while (current_receive_pointer->own_buffer == HOST_BUFFER_OWNERSHIP) {			
1137	/*			
1138	* Local register pointer.			
1139	*/			
1140	rcv = current_receive_pointer;			
1141	/*			
1142	* Check for errors that occurred during reception.			
1143	*/			
1144	if (recv->error_summary == 1) {			
1145	if (recv->framing_error == 1) {			
1146	*error  = FRAMING_ERROR;			
1147	/* outputRoutine("FE"); /* Framing Error */			
1148	}			
1149	if (recv->overflow_error == 1) {			
1150	*error  = OVERFLOW_ERROR;			
1151	/* outputRoutine("OF"); /* Overflow error */			
1152	}			
1153	if (recv->crc_error == 1) {			
1154	*error  = CRC_ERROR;			
1155	/* outputRoutine("CRC"); /* CRC error */			
1156	}			
1157	if (recv->buffer_error == 1) {			
1158	*error  = BUFFER_ERROR;			
1159	/* outputRoutine("BU"); /* Buffer error */			
1160	}			
1161	/* Turn the buffer back over to the LANCE. */			
1162	recv->own_buffer = LANCE_BUFFER_OWNERSHIP;			
1163				
1164	if (++current_receive_pointer >= maximum_receive_pointer)			
1165	current_receive_pointer = minimum_receive_pointer;			
1166				
1167	continue;			
1168	}			
1169				
1170	/* Get the address of the buffer. */			
1171	recv_buffer = (char *) ((recv->high_buffer_address << 16)   recv->low_buffer_address);			
1172				
1173	/* Set up pointers to the UDP and IP parts of the buffer. */			
1174	ip_header = (IP_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER));			
1175	udp_header = (UDP_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER));			
1176	lm_header = (LM_HEADER *) (recv_buffer + sizeof(ETHERNET_HEADER) + sizeof(IP_HEADER) + sizeof(UDP_HEADER));			
1177				
1178	/* Reject broadcast, non-UDP, and incorrect port packets. */			
1179				
1180	if (( (u_long *)recv_buffer != 0xffffffff) && ( (u_short *) (recv_buffer+4) != 0xffff) ) {			
1181	if ( ( (u_short *) (recv_buffer+12) == 0x0800 ) && /* type is IP */			
1182				
1183				
1184				
1185	if (ip_header->protocol == 0x11) {			
1186	switch(udp_header->destination_port) {			
1187	case MODELS_ARE_YOU_ALIVE_PORT:			
1188	process_are_you_alive(recv_buffer,			
1189	ip_header, udp_header, lm_header,			
1190	ALIVE_PACKET);			
1191	break;			
1192	case MODELS_DEATHS_PORT:			
1193	/* respond as if are you alive */			
1194	process_are_you_alive(recv_buffer,			
1195	ip_header, udp_header, lm_header,			
1196	DEATH_PACKET);			
1197	outputRoutine("REMOTE RESET");			
1198	/* I think I'm gonna kill myself */			
1199	reset_cpu(REMOTE_RESET);			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89 TIME 4:42:12 pm	PAGE # 11/27
LINE #	SOURCE TEXT			
1200	break;			
1201	case MODELER_ETHERNET_PORT:			
1202	case MODELER_OUT_OF_BAND_PORT:			
1203	process_other_stuff(recv);			
1204	break;			
1205	}			
1206	}			
1207	else			
1208	{			
1209	/*			
1210	** we have a ICMP			
1211	*/			
1212	if( ip_header->protocol == 0x1 )			
1213	{			
1214	icmp_pkt = (ICMP_HEADER *)udp_header;			
1215	/*			
1216	** check for echo request			
1217	*/			
1218	if( icmp_pkt->type == 8 )			
1219	{			
1220	send_icmp_reply(recv_buffer);			
1221	}			
1222	}			
1223	}			
1224	}			
1225	} else if ( *((u_short *) (recv_buffer+12)) == ARP_ARP ) { /* type is ARP */			
1226	{			
1227	struct ethernet_arp_packet *packet;			
1228	long error;			
1229	packet = (struct ethernet_arp_packet *) recv_buffer;			
1230	if ( packet->arp.arp_opcode == ARP_REPLY && accept_arp_reply ) {			
1231	{			
1232	/* We have received a reply to our ARP			
1233	/* request. Pass it to AI code			
1234	process_other_stuff(recv);			
1235	}			
1236	}			
1237	if ( packet->arp.arp_opcode == ARP_REQUEST ) {			
1238	{			
1239	/* This is an ARP request directed at the			
1240	/* modeler. Here is where we should			
1241	/* initiate a response.			
1242	/*			
1243	if( !is_valid_bvaran == TRUE )			
1244	{			
1245	send_arp_reply ((struct ethernet_arp_packet *) recv_buffer,			
1246	(u_char *) id_prom.ethernet,			
1247	boot.modeler_internet_address,			
1248	packet->arp.arp_source_hardware_address,			
1249	packet->arp.arp_source_protocol_address );			
1250	}			
1251	}			
1252	}			
1253	}			
1254	}			
1255	} else {			
1256	/* We have a broadcast packet which might be of interest */			
1257	{			
1258	struct ethernet_arp_packet *packet;			
1259	packet = (struct ethernet_arp_packet *) recv_buffer;			
1260	if ( packet->arp.arp_opcode == ARP_REQUEST ) {			
1261	{			
1262	if ( packet->arp.target_protocol_address == boot.modeler_internet_address ) {			
1263	{			
1264	/* This is a broadcasted ARP request for			
1265	/* this modeler. Here is where we should			
1266	/* initiate a response.			
1267	/*			
1268	if( !is_valid_bvaran == TRUE )			
1269	{			
1270	send_arp_reply ((struct ethernet_arp_packet *) recv_buffer,			
1271	(u_char *) id_prom.ethernet,			
1272	boot.modeler_internet_address,			
1273	packet->arp.arp_source_hardware_address,			
1274	packet->arp.arp_source_protocol_address );			
1275	}			
1276	}			
1277	}			
1278	}			
1279	}			
1280	too_many_buffers = 0;			
1281	/*			
1282	* Turns the buffer back over to the LANCE.			
1283	*/			
1284	recv->own_buffer = LANCE_BUFFER_OWNERSHIP;			
1285	/*			
1286	* Set our, as opposed to the LANCE's, notion of what the current receive			
1287	* message descriptor is. Actually, this is the next receive descriptor			
1288	* to be used, but I think the code is clearer if it's name is "current".			
1289	*/			
1290	/* The receive message descriptors are in a ring structure which causes			
1291	/* us to have to wrap around to the beginning when the end of the list			
1292	/* is reached.			
1293	*/			
1294	if ( ++current_receive_pointer >= maximum_receive_pointer )			
1295	current_receive_pointer = minimum_receive_pointer;			
1296	}			
1297	}			
1298	}			
1299	}			
1300	static void			
1301	process_are_you_alive(message, ip_header, udp_header, lm_header, flag)			
1302	{			
1303	register char *message;			
1304	register IP_HEADER *ip_header;			
1305	register UDP_HEADER *udp_header;			
1306	register LM_HEADER *lm_header;			
1307	{			
1308	SOCKET_ADDRESS from;			
1309	register ETHERNET_HEADER *eth_header;			
1310	long *lptr = (long *) (lm_header + 1);			
1311	long cmd;			
1312	long size;			
1313	char *buffer;			
1314	{			
1315	extern char modeler_state;			
1316	{			
1317	cmd = *lptr++;			
1318	size = *lptr++;			
1319	{			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 12/28
LINE #	SOURCE TEXT			
1320	buffer = (char *)lptr;			
1321	reset_header = (ETHERNET_HEADER *) message;			
1322	from.packet_type = reset_header->type;			
1323	from.destination[0] = reset_header->source[0];			
1324	from.destination[1] = reset_header->source[1];			
1325	from.destination[2] = reset_header->source[2];			
1326	from.destination[3] = reset_header->source[3];			
1327	from.destination[4] = reset_header->source[4];			
1328	from.destination[5] = reset_header->source[5];			
1329	from.family = AF_INET;			
1330	from.port = udp_header->source_port;			
1331	from.address = ip_header->source_address;			
1332	alive_inet = ip_header->destination_address;			
1333	lm_header->byte_count = sizeof(LM_HEADER) + 4; /* Choose to send 4 bytes. */			
1334	/* All other lm_header fields are already properly set. */			
1335	/* NOTE: update the version when either "unknown" changes to a known. */			
1336	/* At the end of the header, put a version and the desired return data. */			
1337	*(u_char *)((char *)lm_header+sizeof(LM_HEADER)+0) = (u_char) 0x1; /* version */			
1338	*(u_char *)((char *)lm_header+sizeof(LM_HEADER)+1) = (u_char) modeler_state;			
1339	*(u_char *)((char *)lm_header+sizeof(LM_HEADER)+2) = (u_long) 0x0; /* unknown */			
1340	*(u_char *)((char *)lm_header+sizeof(LM_HEADER)+3) = (u_long) 0x0; /* unknown */			
1341	(void) sendto(0, message, (int)lm_header->byte_count,			
1342	UDP_IP_PACKET flag, (struct sockaddr *)&from, SOCK_SIZE);			
1343	switch (cmd) {			
1344	case AYA_INQUIRE:			
1345	break;			
1346	case AYA_KILL:			
1347	outputRoutine("REMOTE RESET");			
1348	reset_cpu(REMOTE_RESET);			
1349	break;			
1350	case AYA_INTERRUPT:			
1351	lm_intr_requested = 1;			
1352	break;			
1353	}			
1354	static void			
1355	process_other_stuff(recv)			
1356	register RECV_MDE *recv;			
1357	{			
1358	int err;			
1359	/*			
1360	* Inform task of packet arrival and let the ISR know			
1361	* we made a system call so it uses UI_EXIT to leave ISR.			
1362	*/			
1363	if (copy_from_lance_ring(recv) == FAILURE) {			
1364	too_many_buffers = 1;			
1365	outputRoutine("WR"); /* No Buffers */			
1366	return; /* non-fatal error */			
1367	}			
1368	system_call = 1;			
1369	if (extra_rcv_desc_count == 1) {			
1370	so_spost (rcv_lance_pkt_semaphore, &err);			
1371	if (err) {			
1372	/* Turn the buffer back over to the LANCE. */			
1373	recv->own_buffer = LANCE_BUFFER_OWNERSHIP;			
1374	if (++current_receive_pointer >= maximum_receive_pointer)			
1375	current_receive_pointer = minimum_receive_pointer;			
1376	outputRoutine("SP"); /* Semaphore Post */			
1377	return; /* non-fatal error */			
1378	}			
1379	}			
1380	static void			
1381	memory_copy(s1, s2, count)			
1382	register u_long *s1;			
1383	register u_long *s2;			
1384	register short count;			
1385	{			
1386	register short remain = count & 3;			
1387	/*			
1388	* divide by 4.			
1389	* How many long moves we need to perform			
1390	*/			
1391	count >>= 2;			
1392	while(--count >= 0) {			
1393	*s1++ = *s2++;			
1394	}			
1395	/*			
1396	* clean up, by moving remaining bytes			
1397	*/			
1398	while(--remain >= 0) {			
1399	*(u_char *)s1++ = *(u_char *)s2++;			
1400	}			
1401	}			
1402	/*			
1403	* LANCE Interrupt Service Routine			
1404	*/			
1405	void			
1406	lance_isr()			
1407	{			
1408	u_short error;			
1409	u_short reset = 0;			
1410	u_short not_a_register_err0;			
1411	register u_short err0;			
1412	register TRANS_MDE *trans_ptr;			
1413	{			
1414	/*			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89 TIME 4:42:12 pm	PAGE # 13/29
SOURCE TEXT				
LINE #				
1440	/* Check the status of LANCE control and status Register 0			
1441	* for possible errors and the source of the interrupt.			
1442	*/			
1443	if (lance_read( &not_a_register_car0 ) != SUCCESS) {			
1444	reset = 1;			
1445	outputRoutine("CSR");			
1446	not_a_register_car0 = 0x0;			
1447	/* I know options are gross, but we don't want to			
1448	* check the stuff below if the CSR read failed.			
1449	*/			
1450	goto close_up_shop;			
1451	}			
1452				
1453				
1454	/*			
1455	* Create a local register variable for car0.			
1456	*/			
1457	car0 = not_a_register_car0;			
1458				
1459	/*			
1460	** If we didn't get an interrupt, just harmlessly return.			
1461	** In reality, something is seriously broken - oh well.			
1462	**			
1463	if((car0 & INTERRUPT_SUMMARY) != INTERRUPT_SUMMARY)			
1464	return;			
1465				
1466	/*			
1467	* Check all the conditions that could have occurred.			
1468	* Note that there are several fatal error conditions.			
1469	*/			
1470	if ((car0 & ERROR_SUMMARY) == ERROR_SUMMARY) {			
1471	/* Note that we don't reset here because if the error is			
1472	* fatal the receiver and/or transmitter will be off and			
1473	* checked below.			
1474	*/			
1475	if ((car0 & TRANSMIT_BABBLE_ERROR) == TRANSMIT_BABBLE_ERROR)			
1476	outputRoutine("BAB");			
1477	if ((car0 & MISSED_PACKET_ERROR) == MISSED_PACKET_ERROR)			
1478	{			
1479	/* outputRoutine("MISS"); */			
1480				
1481				
1482	if ((car0 & MEMORY_ERROR) == MEMORY_ERROR)			
1483	{			
1484	/* outputRoutine("MEM"); */			
1485				
1486				
1487	/* COLLISION_ERROR			
1488	* According to the LANCE technical manual, P4-15, collision error			
1489	* should not be considered fatal. Comment out this test for			
1490	* now but give it a try during system debug/test.			
1491	*/			
1492	if ((car0 & COLLISION_ERROR) == COLLISION_ERROR)			
1493	{			
1494	/* outputRoutine("COL"); */			
1495				
1496				
1497				
1498				
1499	if ((car0 & RECEIVE_ON) != RECEIVE_ON) {			
1500	reset = 1;			
1501	outputRoutine("RX");			
1502				
1503				
1504	if ((car0 & TRANSMIT_ON) != TRANSMIT_ON) {			
1505	reset = 1;			
1506	outputRoutine("TX");			
1507				
1508				
1509	if ( ((car0 & RECEIVE_INTERRUPT) == RECEIVE_INTERRUPT)			
1510	(too_many_buffers && (extra_rcv_desc_count != MAX_USERS)) )			
1511	{			
1512	lance_lar_receive( &error );			
1513				
1514				
1515	if ((car0 & TRANSMIT_INTERRUPT) == TRANSMIT_INTERRUPT) {			
1516	trans_ptr = lance_send_packet;			
1517	if ((trans_ptr->max_packet >= maximum_transmit_pointer) {			
1518	lance_send_packet = minimum_transmit_pointer;			
1519	}			
1520	if(trans_ptr->own_buffer == HOST_BUFFER_OWNERSHIP) {			
1521	if(trans_ptr->buffer_error == 1)			
1522	outputRoutine("BUFF");			
1523	if(trans_ptr->error_summary == 1)			
1524	{			
1525	if(trans_ptr->underflow_error == 1)			
1526	{			
1527	#ifdef BROKEN_HARDWARE			
1528	reinitialize_lance = 1;			
1529	((cpu_control_reg_struct *) CPU_CONTROL_REG->control_spare = 1;			
1530	#endif BROKEN_HARDWARE			
1531	outputRoutine("UFLO");			
1532				
1533	if(trans_ptr->late_collision_error == 1)			
1534	{			
1535	/* outputRoutine("LCOL"); */			
1536				
1537				
1538	if(trans_ptr->lost_carrier_error == 1)			
1539	outputRoutine("LCAR");			
1540	if(trans_ptr->retry_error == 1)			
1541	{			
1542	/* outputRoutine("RT"); */			
1543				
1544				
1545				
1546				
1547				
1548				
1549	close_up_shop:			
1550	if(reset == 1) {			
1551	(void)lance_access((char *) &not_a_register_car0, LANCE_CSR_REG, sizeof_LANCE_CSR_REG, MEMORY_WRITE, &error);			
1552	reset_cpu(LANCE_ERROR);			
1553				
1554				
1555				
1556	/*			
1557	** exchange buffers			
1558	** between lance rcv ring & our spare ring			
1559	** This avoids the situation of having to copy buffers in the ISR			



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89 TIME 4:42:12 pm	PAGE # 14/30
LINE #	SOURCE TEXT			
1560	/*			
1561	copy from lance ring( rcv )			
1562	register RECV_MSE *rcv;			
1563	{			
1564	register RECV_MSE *new_rcv = &extra_rcv_desc_buffers[ in_extra_rcv_desc_ptr ];			
1565	/*			
1566	any buffers?			
1567	/*			
1568	if( (available_extra_buffer < 0)    (extra_rcv_desc_count == MAX_USERS);			
1569	return( FAILURE );			
1570	/*			
1571	copy the rcv descriptor			
1572	/*			
1573	new_rcv = *rcv;			
1574	/*			
1575	bump the 'pointer', and the count			
1576	/*			
1577	in_extra_rcv_desc_ptr++;			
1578	in_extra_rcv_desc_ptr = ( MAX_USERS - 1 );			
1579	/*			
1580	extra_rcv_desc_count++;			
1581	/*			
1582	give lance one of our spare buffers			
1583	/*			
1584	rcv->low_buffer_address =			
1585	(u_short) p_la_rcv_buffers_extra[ available_extra_buffer ];			
1586	rcv->high_buffer_address =			
1587	(unsigned) p_la_rcv_buffers_extra[ available_extra_buffer - 1 ] >> 16;			
1588	return( SUCCESS );			
1589	}			
1590	#ifdef ICMP_MESSAGES			
1591	#define PR(number) { int i; for(i=0; i<number; ++i) { (void)printf("%02x", *ptr++); (void)printf("\n"); }			
1592	#define PR(number) { int i; for(i=0; i<number; ++i) { (void)printf("%02x", *ptr++); (void)printf("\n"); }			
1593	static			
1594	print_garbage_header(message)			
1595	char *message;			
1596	{			
1597	register u_char *ptr;			
1598	{			
1599	ptr = (u_char *)message;			
1600	(void)printf("received garbage\n");			
1601	(void)printf("Eset source: "); PR(6);			
1602	(void)printf("Eset destination: "); PR(6);			
1603	(void)printf("Eset type: "); PR(2);			
1604	(void)printf("Eset V.I.L.Type: "); PR(2);			
1605	(void)printf("Eset length: "); PR(2);			
1606	(void)printf("Eset Id.Frag: "); PR(4);			
1607	(void)printf("Eset TTL,Prot: "); PR(2);			
1608	(void)printf("Eset Chksum: "); PR(2);			
1609	(void)printf("Eset source: "); PR(4);			
1610	(void)printf("Eset destination: "); PR(4);			
1611	if (*(u_char *)message + 23) != 0x1 {			
1612	(void)printf("ERROR: unknown protocol(0x%02x)\n", *(ptr+23));			
1613	}			
1614	else {			
1615	(void)printf("ICMP Type,Code: "); PR(2);			
1616	(void)printf("ICMP Chksum: "); PR(2);			
1617	(void)printf("ICMP unused: "); PR(4);			
1618	(void)printf("Eset V.I.L.Type: "); PR(2);			
1619	(void)printf("Eset length: "); PR(2);			
1620	(void)printf("Eset Id.Frag: "); PR(4);			
1621	(void)printf("Eset TTL,Prot: "); PR(2);			
1622	(void)printf("Eset Chksum: "); PR(2);			
1623	(void)printf("Eset source: "); PR(4);			
1624	(void)printf("Eset destination: "); PR(4);			
1625	}			
1626	#endif ICMP_MESSAGES			
1627	send_icmp_reply(rcv_buf)			
1628	register char *rcv_buf;			
1629	{			
1630	extern ID_PROM_CPU id_prom;			
1631	u_long inet_address, error;			
1632	register PACKET_HEADER *rcv_buffer = (PACKET_HEADER *)rcv_buf;			
1633	register u_long i, csum;			
1634	register u_short j;			
1635	register u_char *enet_address = (u_char *) id_prom.ethernet;			
1636	register ICMP_HEADER *icmp_pkt = (ICMP_HEADER *) (rcv_buf + sizeof( ETHERNET_HEADER ) + sizeof( IP_HEADER ));			
1637	register u_short *icmp_checksum = (u_short *) (rcv_buf + sizeof( ETHERNET_HEADER ) + sizeof( IP_HEADER ));			
1638	/*			
1639	xchg source destination ethernet addresses			
1640	/*			
1641	rcv_buffer->ethernet_hdr.destination[0]=rcv_buffer->ethernet_hdr.source[0];			
1642	rcv_buffer->ethernet_hdr.destination[1]=rcv_buffer->ethernet_hdr.source[1];			
1643	rcv_buffer->ethernet_hdr.destination[2]=rcv_buffer->ethernet_hdr.source[2];			
1644	rcv_buffer->ethernet_hdr.destination[3]=rcv_buffer->ethernet_hdr.source[3];			
1645	rcv_buffer->ethernet_hdr.destination[4]=rcv_buffer->ethernet_hdr.source[4];			
1646	rcv_buffer->ethernet_hdr.destination[5]=rcv_buffer->ethernet_hdr.source[5];			
1647	rcv_buffer->ethernet_hdr.source[0]=enet_address[0];			
1648	rcv_buffer->ethernet_hdr.source[1]=enet_address[1];			
1649	rcv_buffer->ethernet_hdr.source[2]=enet_address[2];			
1650	rcv_buffer->ethernet_hdr.source[3]=enet_address[3];			
1651	rcv_buffer->ethernet_hdr.source[4]=enet_address[4];			
1652	rcv_buffer->ethernet_hdr.source[5]=enet_address[5];			
1653	j = rcv_buffer->ip_hdr.total_length - sizeof( IP_HEADER );			
1654	/*			
1655	xchg source destination internet addresses.			
1656	/*			
1657	inet_address = rcv_buffer->ip_hdr.source_address;			
1658	rcv_buffer->ip_hdr.source_address=rcv_buffer->ip_hdr.destination_address;			
1659	rcv_buffer->ip_hdr.destination_address = inet_address;			
1660	/*			
1661	set reply type			
1662	/*			
1663	icmp_pkt->type = 0;			
1664	/*			
1665	calculate ICMP checksum			
1666	/*			
1667	csum = 0;			
1668	/*			
1669	/*			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lance.c	DATE 5/23/89	PAGE # 15/31
LINE #		SOURCE TEXT		
1680		icmp_pkt->checksum = 0;		
1681		for( i = 0; i < 3; i++)		
1682		caum += *icmp_checksum++;		
1683		caum += caum >> 16; /* add in the carry */		
1684		icmp_pkt->checksum = (~caum); /* the one's complement of the checksum */		
1685		lance_transmit(recv_buf, recv_buffer->ip_hdr.total_length		
1686		+ sizeof( ETHERNET_HEADER ), &error);		
1687		)		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lm_ee_access.c	DATE 5/23/89 TIME 4:42:13 pm	PAGE # 1/32
LINE #	SOURCE TEXT			
1	/* SCCS ID: lm_ee_access.c rev 3.1, 4/24/89 at 07:46:54 */			
2	#include "common.h"			
3	#include "mod_def.h"			
4	#include "cpu.h"			
5	#include "magic.h"			
6	#include "pel.h"			
7	#include "lm_rd_wr.h"			
8	#include "eeprom.h"			
9	u_short access_eeprom();			
10	u_short write_eeprom(), read_eeprom();			
11	void disable_eeprom(), enable_eeprom(), erase_all_eeprom();			
12	u_char diag_dab;			
13	u_short			
14	lm_eeprom_access(data_struct, field, ee_num, sizeof_field, option, error)			
15	char *data_struct;			
16	unsigned long field;			
17	u_long ee_num;			
18	unsigned long sizeof_field, option;			
19	unsigned long *error;			
20	{			
21	register PEL *pel_access_reg;			
22	u_short status;			
23	register u_char ee_number = ee_num;			
24	register unsigned char eeprom = (unsigned char) field;			
25	register unsigned short checksum = 0;			
26	register unsigned short temp, i = 0, *ptr = (u_short *) data_struct;			
27	u_char old_eeprom_in_bit;			
28	u_char old_initialize_bit;			
29	*error = NO_NVRAM_ERROR;			
30	pel_access_reg = (PEL *)			
31	(pel_addr(((ee_number & 0x15) >> 3), (ee_number & 7)));			
32	/*			
33	** check for diag dab by toggling eeprom in			
34	** and present should follow.			
35	/*			
36	diag_dab = 0;			
37	old_eeprom_in_bit = pel_access_reg->car.bit.eeprom_in;			
38	pel_access_reg->car.bit.eeprom_in = 1;			
39	if( pel_access_reg->car.bit.present == 0)			
40	{			
41	pel_access_reg->car.bit.eeprom_in = 0;			
42	if( pel_access_reg->car.bit.present == 1)			
43	{			
44	diag_dab = 1;			
45	}			
46	}			
47	pel_access_reg->car.bit.eeprom_in = old_eeprom_in_bit;			
48	old_initialize_bit = pel_access_reg->car.bit.initialize;			
49	pel_access_reg->car.bit.initialize = 0;			
50	pel_access_reg->car.bit.initialize = 1;			
51	/*			
52	** if reading or writing we have valid eeprom			
53	** and eeprom access is within bounds			
54	/*			
55	if(option == MEMORY_READ    option == MEMORY_WRITE)			
56	{			
57	if(field + sizeof_field > (CPU_EEPROM_SIZE))			
58	{			
59	*error = NO_NVRAM;			
60	pel_access_reg->car.bit.initialize = old_initialize_bit;			
61	return( FAILURE );			
62	}			
63	}			
64	/*			
65	** perform read			
66	/*			
67	if( option == MEMORY_READ )			
68	{			
69	/*			
70	** take care of odd reads			
71	/*			
72	if( eeprom & 1 )			
73	{			
74	*(u_char *) ptr = (u_char) (read_eeprom( ee_number, eeprom >> 1, &status) >> 8);			
75	if( status == FAILURE ) {			
76	pel_access_reg->car.bit.initialize = old_initialize_bit;			
77	return( FAILURE );			
78	}			
79	++eeprom;			
80	(u_char *) ++ptr;			
81	--sizeof_field;			
82	}			
83	eeprom >>= 1;			
84	/*			
85	** Anything else to read			
86	/*			
87	if( sizeof_field )			
88	{			
89	/*			
90	** skip odd read at the end			
91	/*			
92	temp = sizeof_field & 0xffff;			
93	if( temp )			
94	{			
95	for(i = 0; i < temp; i+=2)			
96	{			
97	*ptr++ = read_eeprom( ee_number, eeprom, &status );			
98	if( status == FAILURE ) {			
99	pel_access_reg->car.bit.initialize = old_initialize_bit;			
100	return( FAILURE );			
101	}			
102	eeprom ++;			
103	}			
104	}			
105	}			
106	/*			
107	** do odd read at the end			
108	/*			
109	if( sizeof_field & 1 )			
110	{			
111	*(u_char *) ptr = (u_char) (read_eeprom( ee_number, eeprom, &status));			
112	if( status == FAILURE ) {			
113	pel_access_reg->car.bit.initialize = old_initialize_bit;			
114	return( FAILURE );			
115	}			
116	}			
117	}			
118	/*			
119	** Anything else to read			
120	/*			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/lm\_ee\_access.c

DATE 5/23/89  
TIME 4:42:13 pm

PAGE #  
2/33

```

LINE #      SOURCE TEXT
121          }
122          pel_access_reg->car.bit.initialize = old_initialize_bit;
123          return( SUCCESS );
124      }
125      /*
126      ** perform write
127      */
128      if( option == MEMORY_WRITE )
129      {
130          /*
131          ** take care of odd writes
132          */
133          if( eeprom & 1 )
134          {
135              if( write_eeprom( ee_number, eeprom >> 1, (read_eeprom( ee_number, eeprom >> 1 & 0xff, &status)) | ((u_char *) ptr) << 8)
136              ) == FAILURE )
137              {
138                  *error = WRITE_FAILURE;
139                  pel_access_reg->car.bit.initialize = old_initialize_bit;
140                  return( FAILURE );
141              }
142              if( status == FAILURE ) {
143                  pel_access_reg->car.bit.initialize = old_initialize_bit;
144                  return( FAILURE );
145              }
146              ++eeprom;
147              (u_char *) ++ptr;
148              --sizeof_field;
149          }
150          eeprom >>= 1;
151          /*
152          ** Anything else to write
153          */
154          if( sizeof_field )
155          {
156              /*
157              ** skip odd write at the end
158              */
159              temp = sizeof_field & 0xffff;
160              if( temp )
161              {
162                  for( i = 0; i < temp; i += 2 )
163                  {
164                      if( write_eeprom( ee_number, eeprom, *ptr++ ) == FAILURE )
165                      {
166                          *error = WRITE_FAILURE;
167                          pel_access_reg->car.bit.initialize = old_initialize_bit;
168                          return( FAILURE );
169                      }
170                      eeprom ++;
171                  }
172              }
173              /*
174              ** do odd read at the end
175              */
176              if( sizeof_field & 1 )
177              {
178                  if( write_eeprom( ee_number, eeprom, ((read_eeprom( ee_number, eeprom) & 0x1fff) + ((u_char *) ptr) -
179                  == FAILURE )
180                  {
181                      *error = WRITE_FAILURE;
182                      pel_access_reg->car.bit.initialize = old_initialize_bit;
183                      return( FAILURE );
184                  }
185                  if( status == FAILURE ) {
186                      pel_access_reg->car.bit.initialize = old_initialize_bit;
187                      return ( FAILURE );
188                  }
189              }
190          }
191          /*
192          ** This assumes checksum and write count are on a even boundary
193          */
194          if( write_eeprom( ee_number, EEPROM_WRITE_COUNT, read_eeprom( ee_number, EEPROM_WRITE_COUNT ) & 1, &status) == FAILURE )
195          {
196              *error = WRITE_FAILURE;
197              pel_access_reg->car.bit.initialize = old_initialize_bit;
198              return( FAILURE );
199          }
200          if( status == FAILURE ) {
201              pel_access_reg->car.bit.initialize = old_initialize_bit;
202              return ( FAILURE );
203          }
204          /*
205          ** calculate checksum, do not use checksum in the calculation
206          ** 3's complement of the sum of the rest of the fields
207          */
208          for( i = 0; i <= 63; i++)
209          {
210              checksum += read_eeprom( ee_number, i, &status);
211              if( status == FAILURE ) {
212                  pel_access_reg->car.bit.initialize = old_initialize_bit;
213                  return ( FAILURE );
214              }
215          }
216          checksum = ~checksum;
217          ++checksum;
218          if( write_eeprom( ee_number, EEPROM_CHECKSUM, checksum) == FAILURE )
219          {
220              *error = WRITE_FAILURE;
221              pel_access_reg->car.bit.initialize = old_initialize_bit;
222              return( FAILURE );
223          }
224          pel_access_reg->car.bit.initialize = old_initialize_bit;
225          return( SUCCESS );
226      }
227      /*
228      ** Initialize Sram
229      */
230      if( option == MEMORY_INIT )
231      {
232          /*
233          ** clear out Eeprom
234          */
235          erase_all_eeprom( ee_number );
236          /*
237          ** Fill fields with 0
238          */
239          for( i = EEPROM_SIGNATURE_1 & 1; i < EEPROM_WRITE_COUNT; ++i)

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lm_ee_access.c	DATE * 5/23/89 TIME 4:42:13 pm	PAGE # 3/34
LINE #	SOURCE TEXT			
239	/* read */ write_eepram(ee_number, 1, (u_short) 0);			
240	/* set up signature			
241	*/			
242	if( write_eepram(ee_number, EEPROM_SIGNATURE_0, (u_short) SIG_WORD_0) == FAILURE )			
243	{			
244	/*error = WRITE_FAILURE;			
245	pel_access_reg->car.bit.initialize = old_initialize_bit;			
246	return( FAILURE );			
247	}			
248	/*			
249	/* set up signature			
250	*/			
251	if( write_eepram(ee_number, EEPROM_SIGNATURE_1, (u_short) SIG_WORD_1) == FAILURE )			
252	{			
253	/*error = WRITE_FAILURE;			
254	pel_access_reg->car.bit.initialize = old_initialize_bit;			
255	return( FAILURE );			
256	}			
257	/*			
258	/* This assumes checksum and write count are on a even boundary			
259	*/			
260	if( write_eepram(ee_number, EEPROM_WRITE_COUNT, (u_short) read_eepram( ee_number, EEPROM_WRITE_COUNT )+1, &status) == FAILURE)			
261	{			
262	/*error = WRITE_FAILURE;			
263	pel_access_reg->car.bit.initialize = old_initialize_bit;			
264	return( FAILURE );			
265	}			
266	/*			
267	/* calculate checksum, do not use checksum in the calculation			
268	/* 2's complement of the sum of the rest of the fields			
269	*/			
270	for( i = 0; i != 63; i++)			
271	{			
272	checksum += read_eepram( ee_number, i, &status);			
273	if( status == FAILURE ) {			
274	/*error = WRITE_FAILURE;			
275	pel_access_reg->car.bit.initialize = old_initialize_bit;			
276	return( FAILURE );			
277	}			
278	}			
279	if( write_eepram(ee_number, EEPROM_CHECKSUM, (u_short) ((~checksum) + 1)) == FAILURE)			
280	{			
281	/*error = WRITE_FAILURE;			
282	pel_access_reg->car.bit.initialize = old_initialize_bit;			
283	return( FAILURE );			
284	}			
285	pel_access_reg->car.bit.initialize = old_initialize_bit;			
286	return( SUCCESS );			
287	}			
288	/*			
289	/* validate NVRAM			
290	*/			
291	if( option == MEMORY_VALIDATE )			
292	{			
293	if( read_eepram( ee_number, EEPROM_SIGNATURE_0, &status ) != SIG_WORD_0)			
294	{			
295	/*error = INVALID_NVRAM;			
296	pel_access_reg->car.bit.initialize = old_initialize_bit;			
297	return( FAILURE );			
298	}			
299	if( status == FAILURE ) {			
300	pel_access_reg->car.bit.initialize = old_initialize_bit;			
301	return( FAILURE );			
302	}			
303	if( read_eepram( ee_number, EEPROM_SIGNATURE_1, &status ) != SIG_WORD_1)			
304	{			
305	/*error = INVALID_NVRAM;			
306	pel_access_reg->car.bit.initialize = old_initialize_bit;			
307	return( FAILURE );			
308	}			
309	if( status == FAILURE ) {			
310	pel_access_reg->car.bit.initialize = old_initialize_bit;			
311	return( FAILURE );			
312	}			
313	/*			
314	/* calculate checksum, do not use checksum in the calculation			
315	/* 2's complement of the sum of the rest of the fields			
316	*/			
317	for( i = 0; i != 64; i++)			
318	{			
319	/*ptr = read_eepram( ee_number, i, &status);			
320	if( status == FAILURE ) {			
321	/*error = WRITE_FAILURE;			
322	pel_access_reg->car.bit.initialize = old_initialize_bit;			
323	return( FAILURE );			
324	}			
325	checksum += *ptr++;			
326	}			
327	if( checksum )			
328	{			
329	/*error = INVALID_NVRAM;			
330	pel_access_reg->car.bit.initialize = old_initialize_bit;			
331	return( FAILURE );			
332	}			
333	pel_access_reg->car.bit.initialize = old_initialize_bit;			
334	return( SUCCESS );			
335	}			
336	pel_acc_reg->car.bit.initialize = old_initialize_bit;			
337	return( FAILURE );			
338	}			
339	/*			
340	u_short			
341	lm_read_eepram( ee_number, dab )			
342	u_long ee_number;			
343	DAB_EEPROM *dab;			
344	{			
345	u_long error;			
346	(void) bzero( (char *) dab, sizeof( DAB_EEPROM ) );			
347	if( lm_eepram_access( (char *) dab, (u_long) 0, ee_number, (u_long) sizeof( DAB_EEPROM ), MEMORY_VALIDATE, &error) == FAILURE)			
348	{			
349	/* Don't initialize the EEPROM on read errors.			
350	lm_eepram_access( (char *) dab, (u_long) 0, ee_number, (u_long) sizeof( DAB_EEPROM ), MEMORY_INIT, &error);			
351	/*			
352	return( FAILURE );			
353	}			
354	return( SUCCESS );			
355	}			
356	/*			
357	char trash = 'L';			
358	u_short			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/lm\_ee\_access.c

DATE 5/23/89 PAGE #  
TIME 4:42:13 pm 4/35

```

LINE # SOURCE TEXT
359 lm_write_eepram( ee_number, dab )
360 register u_long ee_number;
361 register DAB_EEPROM *dab;
362 {
363     u_long error;
364     u_short status;
365     DAB_EEPROM dab2;
366
367     if( lm_e2prom_access( (char *) &dab2, (u_long) 0, ee_number,
368         (u_long) sizeof(DAB_EEPROM), MEMORY_VALIDATE, &error) == FAILURE)
369     {
370         lm_e2prom_access( (char *) &dab2, (u_long) 0, ee_number,
371             (u_long) sizeof(DAB_EEPROM), MEMORY_INIT, &error);
372     }
373
374     dab->signature[ 0 ] = 't';
375     dab->signature[ 1 ] = 'M';
376     dab->signature[ 2 ] = 'S';
377     dab->signature[ 3 ] = 'I';
378
379     dab->write_count = read_eepram( (u_char) ee_number,
380         EEPROM_WRITE_COUNT, &status );
381
382     return( lm_e2prom_access( (char *) dab, (u_long) 0, ee_number,
383         (u_long) sizeof(DAB_EEPROM), MEMORY_WRITE, &error));
384 }
385
386 u_short
387 lm_write_eepram_ins_count( ee_number, count)
388 register u_long ee_number;
389 u_short count;
390 {
391     u_long error;
392     return( lm_e2prom_access( (char *) &count, (u_long) (E2_INS_COUNT), ee_number, (u_long) sizeof(short), MEMORY_WRITE, &error));
393 }
394

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lm_rd_wr.c	DATE 5/23/89	PAGE # 1/36
LINE #	SOURCE TEXT			
1	/* SOCS_ID: lm_rd_wr.c rev 3.1, 4/24/89 at 07:46:57 */			
2	/*			
3	** process_lm_read()			
4	** read modeler memory			
5	*/			
6	#include "common.h"			
7	#include "cpu.h"			
8	#include "lm_rd_wr.h"			
9	#include "mod_err.h"			
10	#include "sparam.h"			
11	#include "id.h"			
12	#include "eprom.h"			
13	#include "uart.h"			
14	extern u_short lm_cpuram_access();			
15	extern u_short lm_sparam_access();			
16	extern u_short lm_eprom_access();			
17	extern u_short lm_idprom_access();			
18	extern u_short lm_modeler_access();			
19	extern u_short lm_cpureg_access();			
20	extern u_short lm_duarta_access();			
21	extern u_short lm_duartb_access();			
22	static char init = 0;			
23	static u_short (*rd_wr_routines[ LM_MAX_MEMORY ]());			
24	static void			
25	init_lm_rd_wr()			
26	{			
27	rd_wr_routines[ LM_CPURAM_MEMORY ] = lm_cpuram_access;			
28	rd_wr_routines[ LM_NVRAM_MEMORY ] = lm_sparam_access;			
29	rd_wr_routines[ LM_EPROM_MEMORY ] = lm_eprom_access;			
30	rd_wr_routines[ LM_EEPROM_MEMORY ] = lm_idprom_access;			
31	rd_wr_routines[ LM_MODELER_MEMORY ] = lm_modeler_access;			
32	rd_wr_routines[ LM_CPUREG_MEMORY ] = lm_cpureg_access;			
33	rd_wr_routines[ LM_DUARTA_MEMORY ] = lm_duarta_access;			
34	rd_wr_routines[ LM_DUARTB_MEMORY ] = lm_duartb_access;			
35	init = 1;			
36	}			
37	/*			
38	** read from modeler memory space			
39	*/			
40	u_short			
41	proc_lm_rd( memory_type, ee_number, offset, number_of_bytes, buffer, status)			
42	{			
43	register u_long memory_type;			
44	register u_long ee_number;			
45	register u_long offset;			
46	register u_long number_of_bytes;			
47	register char *buffer;			
48	register u_long *status;			
49	{			
50	/*			
51	** Make sure function array is set up			
52	*/			
53	if( init == 0 )			
54	init_lm_rd_wr();			
55	/*			
56	** do error checking			
57	*/			
58	if( memory_type > LM_MAX_MEMORY )			
59	{			
60	*status = INVALID_PARAMETER;			
61	return( FAILURE );			
62	}			
63	if( number_of_bytes > MAX_RD_WR_BUFFER_SIZE )			
64	{			
65	*status = INVALID_PARAMETER;			
66	return( FAILURE );			
67	}			
68	/*			
69	** make the call			
70	*/			
71	if( memory_type == LM_EEPROM_MEMORY )			
72	return((*rd_wr_routines[ LM_EEPROM_MEMORY ])( buffer, offset, ee_number, number_of_bytes, MEMORY_READ, status ));			
73	else			
74	return((*rd_wr_routines[ memory_type ])( buffer, offset, number_of_bytes, MEMORY_READ, status ));			
75	}			
76	/*			
77	** write to modeler memory space			
78	*/			
79	u_short			
80	proc_lm_wr( memory_type, ee_number, offset, number_of_bytes, buffer, status)			
81	{			
82	register u_long memory_type;			
83	register u_long ee_number;			
84	register u_long offset;			
85	register u_long number_of_bytes;			
86	register char *buffer;			
87	register u_long *status;			
88	{			
89	/*			
90	** Make sure function array is set up			
91	*/			
92	if( init == 0 )			
93	init_lm_rd_wr();			
94	/*			
95	** do error checking			
96	*/			
97	if( memory_type > LM_MAX_MEMORY )			
98	{			
99	*status = INVALID_PARAMETER;			
100	return( FAILURE );			
101	}			
102	if( number_of_bytes > MAX_RD_WR_BUFFER_SIZE )			
103	{			
104	*status = INVALID_PARAMETER;			
105	return( FAILURE );			
106	}			
107	}			
108	/*			
109	** Make sure function array is set up			
110	*/			
111	if( init == 0 )			
112	init_lm_rd_wr();			
113	/*			
114	** do error checking			
115	*/			
116	if( memory_type > LM_MAX_MEMORY )			
117	{			
118	*status = INVALID_PARAMETER;			
119	return( FAILURE );			
120	}			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lm_rd_wr.c	DATE 5/23/89 TIME 4:42:14 pm	PAGE # 2/37
LINE #	SOURCE TEXT			
121	/*			
122	/* make the call			
123	*/			
124	if( memory_type == _2PROM_MEMORY )			
125	return( (*rd_wr_routines[ LM_2PROM_MEMORY ])( buffer, offset, ee_number, number_of_bytes, MEMORY_WRITE, status ));			
126	else			
127	return( (*rd_wr_routines[ memory_type ])( buffer, offset, number_of_bytes, MEMORY_WRITE, status ));			
128	}			
129	/*			
130	/* access CPU RAM			
131	*/			
132	u_short lm_cpurn_access( buffer, offset, number_of_bytes, option, status )			
133	register u_long option;			
134	register u_long offset;			
135	register u_long number_of_bytes;			
136	register char *buffer;			
137	register u_long *status;			
138	{			
139	register char *src_ptr, *dst_ptr;			
140	{			
141	if((offset + number_of_bytes) >= CPU_RAM_SIZE)			
142	{			
143	*status = NO_MEMORY;			
144	return( FAILURE );			
145	}			
146	/*			
147	/* set up pointers			
148	*/			
149	if( option == MEMORY_READ )			
150	{			
151	src_ptr = (char *) (CPU_RAM + offset);			
152	dst_ptr = buffer;			
153	}			
154	if( option == MEMORY_WRITE )			
155	{			
156	dst_ptr = (char *) (CPU_RAM + offset);			
157	src_ptr = buffer;			
158	}			
159	/*			
160	/* Do the copy			
161	*/			
162	while( number_of_bytes-- )			
163	*dst_ptr++ = *src_ptr++;			
164	return( SUCCESS );			
165	}			
166	}			
167	/*			
168	/* access CPU eeprom			
169	*/			
170	u_short lm_eprom_access( buffer, offset, number_of_bytes, option, status )			
171	register u_long option;			
172	register u_long offset;			
173	register u_long number_of_bytes;			
174	register char *buffer;			
175	register u_long *status;			
176	{			
177	register char *src_ptr, *dst_ptr;			
178	register eeprom_struct *eprom;			
179	{			
180	if( option == MEMORY_WRITE )			
181	{			
182	*status = INVALID_PARAMETER;			
183	return( FAILURE );			
184	}			
185	eprom = (eeprom_struct *) CPU_EPROM;			
186	if((offset + number_of_bytes) >= (CPU_EPROM + eeprom->eprom_size))			
187	{			
188	*status = NO_MEMORY;			
189	return( FAILURE );			
190	}			
191	/*			
192	/* set up pointers			
193	*/			
194	if( option == MEMORY_READ )			
195	{			
196	src_ptr = (char *) (CPU_EPROM + offset);			
197	dst_ptr = buffer;			
198	}			
199	/*			
200	/* Do the copy			
201	*/			
202	while( number_of_bytes-- )			
203	*dst_ptr++ = *src_ptr++;			
204	return( SUCCESS );			
205	}			
206	}			
207	/*			
208	/* access CPU idprom			
209	*/			
210	u_short lm_idprom_access( buffer, offset, number_of_bytes, option, status )			
211	register u_long option;			
212	register u_long offset;			
213	register u_long number_of_bytes;			
214	register char *buffer;			
215	register u_long *status;			
216	{			
217	register char *src_ptr, *dst_ptr;			
218	extern id_prom_cpu id_prom;			
219	{			
220	if( option == MEMORY_WRITE )			
221	{			
222	*status = INVALID_PARAMETER;			
223	return( FAILURE );			
224	}			
225	if((offset + number_of_bytes) >= CPU_ID_PROM_SIZE)			
226	{			
227	*status = NO_MEMORY;			
228	return( FAILURE );			
229	}			
230	/*			
231	/* set up pointers			
232	*/			
233	while( number_of_bytes-- )			
234	*dst_ptr++ = *src_ptr++;			
235	return( SUCCESS );			
236	}			
237	}			
238	/*			
239	/*			
240	/*			



Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/lm\_rd\_wr.c

DATE 5/23/89  
TIME 4:42:14 pm

PAGE #  
3/38

```

LINE #      SOURCE TEXT
241      if( option == MEMORY_READ )
242      {
243          arc_ptr = ((char *) &id_prom) + offset;
244          *--arc_ptr = buffer;
245      }
246
247      /*
248      ** Do the copy
249      */
250      while( number_of_bytes-- )
251          *dst_ptr++ = *arc_ptr++;
252
253      return( SUCCESS );
254  }
255
256  /*
257  ** access modeler
258  */
259  u_short lm_modeler_access( buffer, offset, number_of_bytes, option, status )
260  register u_long option;
261  register u_long offset;
262  register u_long number_of_bytes;
263  register char *buffer;
264  register u_long *status;
265  {
266      register char *arc_ptr, *dst_ptr;
267
268      /*
269      ** set up pointers
270      */
271      if( option == MEMORY_READ )
272      {
273          arc_ptr = (char *) (CPU_RAM + offset);
274          dst_ptr = buffer;
275      }
276
277      if( option == MEMORY_WRITE )
278      {
279          dst_ptr = (char *) (CPU_RAM + offset);
280          arc_ptr = buffer;
281      }
282
283      /*
284      ** Do the copy
285      */
286      while( number_of_bytes-- )
287          *dst_ptr++ = *arc_ptr++;
288
289      return( SUCCESS );
290  }
291
292  /*
293  ** access CPUreg
294  ** At present this sets the PC.
295  */
296  u_short lm_cpureg_access( buffer, offset, number_of_bytes, option, status )
297  register u_long option;
298  register u_long offset;
299  register u_long number_of_bytes;
300  register char *buffer;
301  register u_long *status;
302  {
303      register void (*execution_address)();
304      unsigned long err;
305      char mod_state;
306
307      if( offset >= CPU_RAM_SIZE )
308      {
309          *status = NO_MEMORY;
310          return( FAILURE );
311      }
312
313      /*
314      ** set up pointers
315      */
316      if( option == MEMORY_READ )
317      {
318          *status = INVALID_PARAMETER;
319          return( FAILURE );
320      }
321
322      if( option == MEMORY_WRITE )
323      {
324          if( lm_svarm_access( &mod_state, MODELER_STATE, sizeof( MODELER_STATE ), MEMORY_READ, &err ) == FAILURE )
325          {
326              sys_out( "\nFailed to read NVarM state");
327          }
328          if( mod_state != BOOTED )
329          {
330              (void)disable_cache();
331              execution_address = (void *) (offset);
332              (*execution_address)();
333          }
334          return( FAILURE );
335      }
336
337      /*
338      ** access DUART A
339      ** At present this sets the DUART.
340      */
341      u_short lm_duarta_access( buffer, offset, number_of_bytes, option, status )
342      register u_long option;
343      register u_long offset;
344      register u_long number_of_bytes;
345      register char *buffer;
346      register u_long *status;
347      {
348          if( number_of_bytes != 1 )
349          {
350              *status = INVALID_PARAMETER;
351              return( FAILURE );
352          }
353          switch( *buffer & 0xf )
354          {
355              case BAUD_110: /* Baud rate 110 */
356              case BAUD_300: /* Baud rate 300 */
357              case BAUD_1200: /* Baud rate 1200 */
358              case BAUD_2400: /* Baud rate 2400 */
359              case BAUD_4800: /* Baud rate 4800 */
360              case BAUD_9600: /* Baud rate 9600 */
361                  break;
362              default:
363                  *status = INVALID_PARAMETER;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/lm_rd_wr.c	DATE 5/23/89	PAGE # 4/39
LINE #	SOURCE TEXT			
361	return( FAILURE );			
362	/*			
363	/* set up pointers			
364	*/			
365	if( option == MEMORY_READ )			
366	{			
367	*status = INVALID_PARAMETER;			
368	return( FAILURE );			
369	/*			
370	/* set baud rate			
371	*/			
372	if( option == MEMORY_WRITE )			
373	{			
374	CPU_DISABLE_INTERRUPTS;			
375	uart_init( "buffer );			
376	CPU_ENABLE_INTERRUPTS;			
377	}			
378	return( SUCCESS );			
379	/*			
380	/* access DUART B			
381	/* At present this sets the DUART.			
382	*/			
383	u_short lm_duart_access( buffer, offset, number_of_bytes, option, status )			
384	register u_long option;			
385	register u_long offset;			
386	register u_long number_of_bytes;			
387	register char *buffer;			
388	register u_long *status;			
389	{			
390	if( number_of_bytes != 1 )			
391	{			
392	*status = INVALID_PARAMETER;			
393	return( FAILURE );			
394	}			
395	switch( *buffer & 0xf )			
396	{			
397	case BAUD_110: /* Baud rate 110 */			
398	case BAUD_300: /* Baud rate 300 */			
399	case BAUD_1200: /* Baud rate 1200 */			
400	case BAUD_2400: /* Baud rate 2400 */			
401	case BAUD_4800: /* Baud rate 4800 */			
402	case BAUD_9600: /* Baud rate 9600 */			
403	break;			
404	default:			
405	*status = INVALID_PARAMETER;			
406	return( FAILURE );			
407	}			
408	/*			
409	/* set up pointers			
410	*/			
411	if( option == MEMORY_READ )			
412	{			
413	*status = INVALID_PARAMETER;			
414	return( FAILURE );			
415	/*			
416	/* set baud rate			
417	*/			
418	if( option == MEMORY_WRITE )			
419	{			
420	CPU_DISABLE_INTERRUPTS;			
421	uart_init( "buffer );			
422	CPU_ENABLE_INTERRUPTS;			
423	}			
424	return( SUCCESS );			
425	/*			
426	/* get vvarm bauda(baud)			
427	unsigned long *baud;			
428	{			
429	unsigned long err;			
430	if( !vvarm_access( baud, BAUD_RATEA, sizeof_BAUD_RATEA,			
431	MEMORY_READ, &err ) == SUCCESS ) {			
432	*baud >= 24;			
433	} else {			
434	*baud = BAUD_96_12_24;			
435	}			
436	/*			
437	/* get vvarm baudb(baud)			
438	unsigned long *baud;			
439	{			
440	unsigned long err;			
441	if( !vvarm_access( baud, BAUD_RATEB, sizeof_BAUD_RATEB,			
442	MEMORY_READ, &err ) == SUCCESS ) {			
443	*baud >= 24;			
444	} else {			
445	*baud = BAUD_24_96_12;			
446	}			
447	/*			
448	/*			
449	/*			
450	/*			
451	/*			
452	/*			
453	/*			
454	/*			
455	/*			
456	/*			
457	/*			

Copyright 1989

Logic Modeling Systems

SOURCE PROGRAM

os/malloc.c

DATE 5/23/89

PAGE #

TIME 4:42:14 pm

1/40

```

1  // SCCS_ID: malloc.c rev 3.1, 4/24/89 at 07:47:00  */
2
3  /*
4   * Allocator/Deallocator
5   * singly-linked, first-fit
6   * roving pointer, coalescing during allocation
7   * free call reverts roving pointer to newly-freed block
8   */
9
10 #define CHECK
11 #define ACCOUNT
12 #ifdef ACCOUNT
13 extern unsigned long available_malloc_size; /* to track remaining memory */
14 #endif ACCOUNT
15 extern int malloc_semaphore;
16 static int err;
17 static unsigned long rover = 0;
18
19 void
20 create_malloc_partition ( start, size )
21 register char *start;
22 register unsigned long size;
23 { register unsigned long addr, end_addr;
24 #ifdef CHECK
25     if ( rover )
26         ( void ) printf ( "create called more than once\n" );
27     return;
28 #endif CHECK
29 #ifdef DEBUG
30     ( void ) printf ( "init(0lx,0lu)\n", ( unsigned long ) start, size );
31 #endif DEBUG
32     addr = ( unsigned long ) start; /* get the start address */
33     end_addr = addr + size; /* point past the arena */
34     addr = 3; /* to round up to 4-byte boundary */
35     end_addr = end_addr + 3; /* clear bottom 2 bits */
36     rover = ( unsigned long ) end_addr; /* point past the arena */
37     rover [ -1 ] = addr + 5; /* zero-length last block is busy */
38     rover = ( unsigned long ) addr; /* initialize rover to first block-1 */
39     /* link first block to last block */
40     rover = end_addr; /* link first block to last block */
41     /* create_malloc_partition */
42
43 static char *
44 allocate ( size )
45 register unsigned long size;
46 { register unsigned long save, next; /* to save our starting point */
47   register unsigned long tmp_size; /* temp to store a block's size */
48 #ifdef CHECK
49     if ( rover == 0 )
50         return printf ( "allocate called before create\n" ), ( char * ) 0;
51     if ( size == 0 )
52         return printf ( "allocate called with zero size\n" ), ( char * ) 0;
53 #endif CHECK
54 #ifdef DEBUG
55     ( void ) printf ( "allo(0lu)= ", size );
56 #endif DEBUG
57     size = 3; /* to round up to a multiple of 4 */
58     size = size + 3; /* clear bottom 2 bits */
59     save = rover; /* so we know where we have seen it all */
60     do /* loop through blocks */
61     { if ( rover [ -1 ] & 1 ) /* block is busy */
62       rover = ( unsigned long ) ( rover [ -1 ] - 1 ); /* to next block */
63     else /* get a free block */
64       while ( ( next = ( unsigned long ) rover [ -1 ] ) & 1 ) /* check for coalescing */
65         while ( ( next [ -1 ] & 1 ) == 0 ) /* repeatedly coalesce */
66           { rover [ -1 ] = next [ -1 ]; /* do the coalesce */
67             next = ( unsigned long ) rover [ -1 ]; /* check next block */
68           }
69       tmp_size =
70         rover [ -1 ] - ( unsigned long ) rover - 4; /* size of block */
71       if ( tmp_size == size ) /* the block is exactly the right size */
72         { save = rover; /* save the pointer to be returned */
73           rover = ( unsigned long ) rover [ -1 ]; /* to next block */
74           ++save [ -1 ]; /* mark this block as busy */
75         }
76 #ifdef ACCOUNT
77         available_malloc_size -= size + 4;
78 #endif ACCOUNT
79 #ifdef DEBUG
80         ( void ) printf ( "[1]0lx\n", ( unsigned long ) save );
81 #endif DEBUG
82         return ( char * ) save; /* there's the block requested */
83     }
84     else if ( tmp_size > size ) /* the block is larger than we need */
85     { size >>= 2; /* change from a byte count to a word count */
86       rover [ size ] = rover [ -1 ]; /* create next block */
87       rover [ -1 ] = ( unsigned long ) ( rover + size ); /* link */
88       save = rover; /* save the pointer to be returned */
89       rover = size; /* bump rover to next block */
90       ++save [ -1 ]; /* mark this block as busy */
91 #ifdef ACCOUNT
92       available_malloc_size -= size << 2;
93 #endif ACCOUNT
94 #ifdef DEBUG
95       ( void ) printf ( "[2]0lx\n", ( unsigned long ) save );
96 #endif DEBUG
97       return ( char * ) save; /* there's the block requested */
98     }
99     else /* the block is too small */
100     rover = ( unsigned long ) rover [ -1 ]; /* to next block */
101 }
102 while ( rover != save ); /* until we come back to where we started */
103 #ifdef DEBUG
104     ( void ) printf ( "[0]NULL\n" );
105 #endif DEBUG
106 return 0; /* failure, no free blocks are large enough */
107 } /* allocate */
108
109 static void
110 deallocate ( ptr )
111 register char *ptr;
112 {
113 #ifdef CHECK
114     register unsigned long addr;
115     if ( ptr == 0 )
116         ( void ) printf ( "NULL pointer passed to deallocate\n" );
117     return;
118 }
119     addr = ( unsigned long ) ptr;

```

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/malloc.c	DATE 5/23/89	PAGE # 2/41
SOURCE TEXT				
LINE #	SOURCE TEXT			
221	if ( addr & 3 )			
222	{ ( void ) printf ( "bad pointer passed to deallocate\n" ) ;			
223	return ;			
224	}			
225	revar = ( unsigned long * ) addr ;			
226	if ( ( revar [ -1 ] & 1 ) == 0 )			
227	{ ( void ) printf ( "free block passed to deallocate\n" ) ;			
228	return ;			
229	}			
230	# endif CHECK			
231	# ifdef DEBOG			
232	{ ( void ) printf ( "free(1k)\n" , ( unsigned long ) ptr ) ;			
233	# endif DEBOG			
234	revar = ( unsigned long * ) ptr , /* turn char pointer into long pointer */			
235	/* revar [ -1 ] , /* mark the passed block as free */			
236	# ifdef ACCOUNT			
237	available_malloc_size -- revar [ -1 ] - ( unsigned long ) revar ;			
238	# endif ACCOUNT			
239	} /* deallocate */			
240	}			
241	char *			
242	malloc ( size )			
243	{ register unsigned long size ;			
244	{ register char *ret ;			
245	{ if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )			
246	return printf ( "ac_spend() failed in malloc()\n" ) , ( char * ) 0 ;			
247	ret = allocate ( size ) ;			
248	{ if ( ac_spend ( malloc_semaphore , terr ) , err )			
249	{ ( void ) printf ( "ac_spend() failed in malloc()\n" ) ;			
250	return ret ;			
251	} /* malloc */			
252	}			
253	void			
254	free ( ptr )			
255	{ register char *ptr ;			
256	{ if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )			
257	{ ( void ) printf ( "ac_spend() failed in free()\n" ) ;			
258	return			
259	{			
260	deallocate ( ptr ) ;			
261	{ if ( ac_spend ( malloc_semaphore , terr ) , err )			
262	{ ( void ) printf ( "ac_spend() failed in free()\n" ) ;			
263	} /* free */			
264	}			
265	char *			
266	realloc ( ptr , size )			
267	{ register char *ptr ;			
268	register unsigned long size ;			
269	{ register char *new , *ret ;			
270	register unsigned long old ;			
271	register unsigned long *rev ;			
272	# ifdef CHECK			
273	if ( ptr == 0 )			
274	{ ( void ) printf ( "NULL pointer passed to realloc()\n" ) ;			
275	return 0 ;			
276	}			
277	# endif CHECK			
278	if ( ac_spend ( malloc_semaphore , 0 , terr ) , err )			
279	return printf ( "ac_spend() failed in realloc()\n" ) , ( char * ) 0 ;			
280	rev = ( unsigned long * ) ptr ;			
281	old = rev [ -1 ] - ( unsigned long ) ptr - 5 ;			
282	deallocate ( ptr ) ;			
283	ret = new = allocate ( size ) ;			
284	if ( new == 0 )			
285	{ ++rev [ -1 ] ;			
286	# ifdef ACCOUNT			
287	available_malloc_size -- old + 4 ;			
288	# endif ACCOUNT			
289	{			
290	else if ( new != ptr )			
291	{ if ( old > size )			
292	old = size ;			
293	while ( old-- )			
294	*new++ = *ptr++ ;			
295	}			
296	{ if ( ac_spend ( malloc_semaphore , terr ) , err )			
297	{ ( void ) printf ( "ac_spend() failed in realloc()\n" ) ;			
298	return ret ;			
299	} /* realloc */			
300	}			
301	char *			
302	calloc ( count , size )			
303	{ register unsigned long count , size ;			
304	{ register char *ret ;			
305	ret = malloc ( count * size ) ;			
306	if ( ret )			
307	bzero ( ret , count ) ;			
308	return ret ;			
309	} /* calloc */			
310				

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/mem_acc.c	DATE 5/23/89	PAGE # 1/42
LINE #	SOURCE TEXT			
1	/* SCCS ID: mem_acc.c rev 3.1. 4/24/89 at 07:47:04 */			
2				
3	#include "common.h"			
4	#include "lm_rd_wr.h"			
5	#include "device.h"			
6	#include "message.h"			
7	#include "hardware.h"			
8	#include "eeprom.h"			
9	#include "laser.h"			
10	#include "protans.h"			
11	#include "lntwork.h"			
12	#include "network.h"			
13				
14	char read_err_buf[] = "Failure to read from the Modeler";			
15	char write_err_buf[] = "Failure to write to the Modeler";			
16				
17	void			
18	process_lm_read( user )			
19	USER_INFO *user;			
20	{			
21	#ifdef MODELER			
22	register u_long memory_type;			
23	register u_long ee_number;			
24	register u_long offset;			
25	register u_long number_of_bytes, i;			
26	register char *err_msg;			
27	char buffer[ MAX_RD_WB_BUFFER_SIZE ];			
28	register char *buf_ptr = buffer;			
29	u_long status;			
30				
31	reset_obuf();			
32	lm_put_int(READ_ANS);			
33	memory_type = lm_get_int();			
34	ee_number = lm_get_int();			
35	offset = lm_get_int();			
36	number_of_bytes = lm_get_int();			
37	/*			
38	read memory, and copy output buffer			
39	*/			
40	if( proc_lm_rd( memory_type, ee_number, offset, number_of_bytes, buffer, &status ) == FAILURE )			
41	{			
42	lm_put_int( 1 ); /* Failure */			
43	lm_put_char( ERROR_MSG );			
44	err_msg = read_err_buf;			
45	while(*err_msg)			
46	lm_put_char( *err_msg++ );			
47	lm_put_char( 0 );			
48	}			
49	else			
50	{			
51	status=0;			
52	lm_put_int( status );			
53	for( i=0; i < number_of_bytes; ++i)			
54	lm_put_char( *buf_ptr++ );			
55	}			
56	end_put(user->id);			
57	return;			
58	/*else			
59	{			
60	reset_obuf();			
61	lm_put_int(READ_ANS);			
62	lm_put_int( 0 ); /* status */			
63	lm_put_int( (u_long) FAILURE );			
64	end_put(user->id);			
65	return;			
66	/*endif			
67	{			
68				
69				
70	void			
71	process_lm_write( user )			
72	USER_INFO *user;			
73	{			
74	#ifdef MODELER			
75	register u_long memory_type;			
76	register u_long ee_number;			
77	register u_long offset;			
78	register char *err_msg;			
79	register u_long number_of_bytes, i;			
80	register char buffer[ MAX_RD_WB_BUFFER_SIZE ];			
81	register char *buf_ptr = buffer;			
82	u_long status;			
83				
84	reset_obuf();			
85	lm_put_int(WRITE_ANS);			
86	memory_type = lm_get_int();			
87	ee_number = lm_get_int();			
88	offset = lm_get_int();			
89	number_of_bytes = lm_get_int();			
90	buf_ptr = buffer;			
91	for( i=0; i < number_of_bytes; ++i)			
92	*buf_ptr++ = lm_get_char();			
93	/*			
94	write to memory, and copy status to output			
95	*/			
96	if( proc_lm_wr( memory_type, ee_number, offset, number_of_bytes, buffer, &status ) == FAILURE )			
97	{			
98	lm_put_int(1); /* Failure */			
99	lm_put_char( ERROR_MSG );			
100	err_msg = write_err_buf;			
101	while(*err_msg)			
102	lm_put_char( *err_msg++ );			
103	lm_put_char( 0 );			
104	}			
105	else			
106	{			
107	lm_put_int( 0 );			
108	}			
109	end_put(user->id);			
110	return;			
111	/*endif			
112	{			
113				
114				
115	reset_obuf();			
116	lm_put_int(WRITE_ANS);			
117	lm_put_int( 0 ); /* status */			
118	lm_put_int( (u_long) FAILURE );			
119	end_put(user->id);			
120	return;			

1819

5,353,243

1820

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/mem_acc.c		DATE * 5/23/89	PAGE #
				TIME 4:42:15 pm	2/43
SOURCE TEXT					
LINE #					
121	endif				
122					

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/mod_err.c	DATE 5/23/89 TIME 4:42:15 pm	PAGE # 1/44
LINE #	SOURCE TEXT			
1	/* SCCS ID: mod_err.c rev 3.1.1, 4/24/89 at 07:47:08 */			
2	#include "common.h"			
3	#include "tag.h"			
4	#include "pac.h"			
5	#include "magic.h"			
6	#include "mod_def.h"			
7	#include "pel.h"			
8	#include "mod_err.h"			
9	#include "cpu.h"			
10	#include "avram.h"			
11	#include "la_rd_wr.h"			
12	#include "vrtx.h"			
13	char buffer[256];			
14	LM_HARDWARE_ERROR modeler_error;			
15	PAC *pacptr[] = {			
16	(PAC *) (LANE_A_OFFSET + PAC_REG_OFFSET),			
17	(PAC *) (LANE_B_OFFSET + PAC_REG_OFFSET),			
18	(PAC *) (LANE_C_OFFSET + PAC_REG_OFFSET),			
19	(PAC *) (LANE_D_OFFSET + PAC_REG_OFFSET)			
20	};			
21	u_char play_completed_flag = 0;			
22	extern u_short la_read_probe(), la_avram_access();			
23	extern void reset_cpu(), output_routine();			
24	void disable_mod_err();			
25	extern TMC *tmcptr;			
26	extern u_char post_end_of_play;			
27	extern int play_semaphore;			
28	static char unknown_source_of_interrupt = 0;			
29	void			
30	mod_error_isr()			
31	{			
32	unsigned long error;			
33	int err;			
34	PEL pel_access_reg;			
35	PEL *pel_access_reg_ptr;			
36	u_short junk, j, i;			
37	u_short k, lanes_enable, lanes_short, multi_lanes;			
38	register char fatal = 0, found_source_of_interrupt = 0;			
39	u_char found_pel_error;			
40	/*			
41	** initialize some variables.			
42	*/			
43	lanes_enable = lanes_short = 0;			
44	++play_completed_flag;			
45	/*			
46	** bump count of unknown errors, then set to 0 if source is found.			
47	*/			
48	if((tmcptr->pattern_intr_enable == 1 && post_end_of_play == 1)			
49	{			
50	ac_sport( play_semaphore, ierr );			
51	if( err != VRTX_OK)			
52	{			
53	printf("Error posting in mod_error_isr() status=%x\n",err);			
54	}			
55	}			
56	/*			
57	** did we get a error on the lanes			
58	*/			
59	if( tmcptr->lane_intr == 0)			
60	{			
61	if((tmcptr->backplane_mode == 0 && tmcptr->pattern_intr_enable == 1)			
62	{			
63	/*			
64	** This happens if we get EOP without errors			
65	*/			
66	printf("EOP interrupt without any lanes having fault\n");			
67	/*			
68	tmcptr->pattern_intr_enable = 0;			
69	/*			
70	** EOP interrupt.			
71	*/			
72	found_source_of_interrupt=1;			
73	/*			
74	** we know source.			
75	*/			
76	unknown_source_of_interrupt = 0;			
77	return;			
78	}			
79	}			
80	else {			
81	/*			
82	output_routine("SE\n");			
83	/*			
84	modeler_error.error = 1;			
85	/*			
86	** see if TMC is driving ERROR			
87	*/			
88	if( tmcptr->tmc_intr    tmcptr->backplane_error )			
89	{			
90	/*output_routine("TMC error interrupt\n");*/			
91	modeler_error.tmc_error = 1;			
92	/*			
93	** if tmc error, read and store tmc error latches			
94	*/			
95	if( tmcptr->tmc_intr == 1)			
96	{			
97	/*			
98	** read and store tmc error latches			
99	*/			
100	modeler_error.lane_enable = tmcptr->lane_enable;			
101	/*			
102	** determine if it was a multi lane play			
103	*/			
104	/* since shorts can cause sync problems with multi lane			
105	** play.			
106	*/			
107	lanes_enable = (u_short) modeler_error.lane_enable;			
108	multi_lanes = lanes_enable >> 1;			
109	multi_lanes = (lanes_enable >> 1) & 1;			
110	multi_lanes = (lanes_enable >> 2) & 1;			
111	multi_lanes = (lanes_enable >> 3) & 1;			
112	modeler_error.lane_b_pel_control = tmcptr->lane_b_pel_control;			
113	modeler_error.lane_b_data_valid = tmcptr->lane_b_data_valid;			
114	modeler_error.lane_a_pel_control = tmcptr->lane_a_pel_control;			
115	modeler_error.lane_a_data_valid = tmcptr->lane_a_data_valid;			
116	modeler_error.lane_d_pel_control = tmcptr->lane_d_pel_control;			
117	modeler_error.lane_d_data_valid = tmcptr->lane_d_data_valid;			
118	}			
119	}			
120	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/mod\_err.c

DATE	5/23/89	PAGE #
TIME	4:42:15 pm	2/45

```

SOURCE TEXT
LINE #
121     modeler_error.lane_c_pel_control = tmgptr -> lane_c_pel_control;
122     modeler_error.lane_c_data_valid = tmgptr -> lane_c_data_valid;
123 }
124 tmgptr -> backplane_error = 0;
125 tmgptr -> _intr_clearl = 0;
126
127 if (lm_tmg_clocken() == FAILURE)
128     reset_cpu_on_fatal_error();
129
130 while( i++ < 0x8000 );
131
132 if (lm_tmg_clockoff() == FAILURE)
133     reset_cpu_on_fatal_error();
134
135 tmgptr -> tmg_intr_clearl = 1;
136
137 /*
138 sprintf(buffer, "1: 3d %x\n",
139     tmgptr->tmg_intr, tmgptr->lane_intr);
140 output_routine(buffer);
141 */
142 /*
143 ** which lanes have error
144 */
145 modeler_error.lane_errors |= tmgptr -> lane_intr;
146 /*
147 ** this is a fatal condition
148 */
149 fatal = 1;
150 /*
151 ** found an error
152 */
153 found_source_of_interrupt=1;
154 }
155 i = 0;
156 /*
157 ** Check to see if we are
158 ** stuck in presentation mode
159 */
160 while( tmgptr -> backplane_mode == 1)
161 {
162     /*
163     ** delay a while before trying
164     ** give up if we can not resolve in two tries.
165     */
166     while( i++ < 0x8000 );
167     /*
168     ** go about play
169     */
170     lm_tmg_abort_play();
171     if( i++ > 0x8002 at tmgptr -> backplane_mode == 1)
172     {
173         modeler_error.tmg_error = 1;
174         /*
175         ** save in MVaran
176         */
177         (void)lm_svaran_access((char *) &modeler_error, HARDWARE_ERROR, sizeof(HARDWARE_ERROR), MEMORY_WRITE,&error);
178         reset_cpu(BACKPLANE_ERROR);
179     }
180     /*
181     ** found an error
182     */
183     found_source_of_interrupt=1;
184 }
185
186 /*
187 sprintf(buffer, "2: 3d %x\n",
188     tmgptr->tmg_intr, tmgptr->lane_intr);
189 output_routine(buffer);
190 */
191 /*
192 ** which lanes have error
193 */
194 modeler_error.lane_errors |= tmgptr -> lane_intr;
195 /*
196 ** go check PAC and PEL
197 ** PAC first
198 */
199 for( i=0; i != 4; i++)
200 {
201     /*
202     ** is there any hardware there
203     */
204     if(! ( tmgptr -> lane_intr & ( 1 << i )))
205         continue;
206     if(lm_read_probe( (u_long *) (pacptr[ i ] ) ) == SUCCESS )
207     {
208         /*
209         ** This is fatal refresh error
210         */
211         if( pacptr[ i ] -> refresh_error == 1)
212         {
213             modeler_error.pac_lane_errors |= 1 << i;
214             modeler_error.pac_error[ i ].pac_refresh_error = 1;
215             fatal = 1;
216             /*output_routine("refresh error\n");*/
217         }
218         /*
219         ** This is fatal request error
220         */
221         if( pacptr[ i ] -> request_error == 1)
222         {
223             modeler_error.pac_lane_errors |= 1 << i;
224             modeler_error.pac_error[ i ].pac_request_error = 1;
225             fatal = 1;
226             /*output_routine("request error\n");*/
227         }
228         /*
229         ** This is fatal pattern error
230         */
231         if( pacptr[ i ] -> pattern_error == 1)
232         {
233             modeler_error.pac_lane_errors |= 1 << i;
234             modeler_error.pac_error[ i ].pac_pattern_error = 1;
235             fatal = 1;
236             /*output_routine("pattern error\n");*/
237         }
238         /*
239         ** This is a fatal parity error
240         */

```



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/mod_err.c	DATE 5/23/89 TIME 4:42:15 pm	PAGE # 3/46
LINE #	SOURCE TEXT			
241	if( pacptr( 1 ) -> parity_error == 1 )			
242	{			
243	/*			
244	** The block where error was found			
245	** and branch address			
246	*/			
247	modeler_error.pac_lane_errors  = 1 << 1;			
248	modeler_error.pac_error( 1 ).pac_branch_address = pacptr( 1 ) -> branch_address;			
249	modeler_error.pac_error( 1 ).pac_block_offset = pacptr( 1 ) -> block_offset;			
250	modeler_error.pac_error( 1 ).pac_control_word_parity_error = 1;			
251	fatal = 1;			
252	/*output routine("parity error\n");*/			
253	}			
254	/*			
255	** There was a parity error on high word			
256	*/			
257	if( pacptr( 1 ) -> high_word_parity_error )			
258	{			
259	/*			
260	** save the address where parity error was detected			
261	*/			
262	modeler_error.pac_lane_errors  = 1 << 1;			
263	modeler_error.pac_error( 1 ).pac_parity_error_address = pacptr( 1 ) -> parity_error_address;			
264	modeler_error.pac_error( 1 ).pac_high_word_parity_error = 1;			
265	fatal = 1;			
266	/*output routine(" High word parity error\n");*/			
267	}			
268	/*			
269	** There was a parity error on low word			
270	*/			
271	if( pacptr( 1 ) -> low_word_parity_error )			
272	{			
273	/*			
274	** save the address where parity error was detected			
275	*/			
276	modeler_error.pac_lane_errors  = 1 << 1;			
277	modeler_error.pac_error( 1 ).pac_parity_error_address = pacptr( 1 ) -> parity_error_address;			
278	modeler_error.pac_error( 1 ).pac_low_word_parity_error = 1;			
279	fatal = 1;			
280	/*output routine(" Low word parity error\n");*/			
281	}			
282	/*			
283	** clear the errors, by writing to any bit in the register			
284	*/			
285	pacptr( 1 ) -> pattern_error = 0;			
286	{			
287	for( j = 0; j != 8; j++ )			
288	{			
289	pel_access_reg_ptr = (PEL *) (pel_addr( 1, j ));			
290	if( !is_read_probe( (u_long *) pel_access_reg_ptr ) == FAILURE )			
291	continue;			
292	pel_access_reg.car.reg = (u_short) pel_access_reg_ptr -> car.reg;			
293	/*			
294	** Check out the PEL			
295	*/			
296	found_pel_error = FALSE;			
297	if( pel_access_reg.car.bit.error1 == 0 )			
298	{			
299	/*			
300	sprintf( buffer, "T00411\n", pel_access_reg.car.reg );			
301	outputRoutine( buffer );			
302	/*			
303	if( pel_access_reg.car.bit.present == 1 )			
304	{			
305	/*			
306	** DAB is present			
307	*/			
308	if( pel_access_reg.car.bit.active == 0 )			
309	{			
310	/*			
311	** DAB plugged in but not active			
312	*/			
313	printf( "DAB inserted\n" );			
314	outputRoutine( "I" );			
315	/*			
316	modeler_error.dab_change = 1;			
317	modeler_error.pel_error( 1 * 8 + j ).dab_inserted = 1;			
318	found_pel_error = TRUE;			
319	}			
320	}			
321	/*			
322	** DAB is absent			
323	*/			
324	if( pel_access_reg.car.bit.initialize == 1 )			
325	{			
326	/*			
327	** DAB was plugged in & set active			
328	*/			
329	printf( "DAB removed\n" );			
330	outputRoutine( "Y" );			
331	/*			
332	lanes_short  = (u_short) 1 << 1;			
333	modeler_error.dab_change = 1;			
334	modeler_error.pel_error( 1 * 8 + j ).dab_removed = 1;			
335	found_pel_error = TRUE;			
336	}			
337	}			
338	/*			
339	** play error, either DAB not present or			
340	** PEL not active.			
341	*/			
342	if( pel_access_reg.car.bit.play_error1 == 0 )			
343	{			
344	/*			
345	output routine("Play error\n");*/			
346	lanes_short  = (u_short) 1 << 1;			
347	modeler_error.pel_error( 1 * 8 + j ).play_error = 1;			
348	found_pel_error = TRUE;			
349	}			
350	if( pel_access_reg.car.bit.magic_error1 == 0 )			
351	{			
352	for( k=0; k < 5; ++k )			
353	{			
354	/*			
355	** check parity			
356	*/			
357	}			
358	}			
359	/*			
360	**			

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM os/mod_err.c	# DATE = 5/23/89 TIME 4:42:15 pm	PAGE # 4/47
LINE #	SOURCE TEXT		
361	if( pel_access_reg_ptr->magic_chip[ k ].m.parity_out != 0 )		
362	{		
363	/*output routine "pel parity error"*/		
364	modeler_error.pel_error[ i = 8 + j ].any_magic_parity = 1;		
365	modeler_error.pel_error[ i = 8 + j ].magic_parity_out  = 1 << k;		
366	if( !m_read_probe( u_long *) (pacptr[ i ]) ) == SUCCESS )		
367	{		
368	modeler_error.pac_error[ i ].pac_branch_address = pacptr[ i ] -> branch_address;		
369	modeler_error.pac_error[ i ].pac_block_offset = pacptr[ i ] -> block_offset;		
370	fatal = 1;		
371	}		
372	/*		
373	** check shorts		
374	*/		
375	if( (junk = pel_access_reg_ptr->magic_chip[ k ].m.short_sample) != 0 )		
376	{		
377	/*output routine "pel short error"*/		
378	lases_short  = (u_short) 1 << i;		
379	modeler_error.pel_error[ i = 8 + j ].any_magic_short = 1;		
380	modeler_error.pel_error[ i = 8 + j ].magic_short[ k ] = junk;		
381	}		
382	}		
383	/*		
384	** reset MAGIC chip		
385	*/		
386	junk = pel_access_reg_ptr->magic_chip[ 0 ].m.reset;		
387	found_pel_error = TRUE;		
388	{		
389	/*		
390	** do reset PEL		
391	*/		
392	if (found_pel_error == TRUE) {		
393	{		
394	/*		
395	sprintf(buffer, "RST: %d ad %n", i, j);		
396	outputRoutine(buffer);		
397	/*		
398	Reset the PEL only if we really find		
399	the source of interrupt.		
400	*/		
401	pel_access_reg_ptr->car.bit.reset=0;		
402	/*		
403	found source of error, error with PEL		
404	*/		
405	modeler_error.pel_error_list  = 1 << ( i = 8 + j );		
406	}		
407	else {		
408	/*		
409	Toggle Reset to clear the PEL error */		
410	pel_access_reg_ptr->car.bit.reset=0;		
411	pel_access_reg_ptr->car.bit.reset=1;		
412	}		
413	found_source_of_interrupt=1;		
414	}		
415	}		
416	if( !tagptr -> lase_intr )		
417	{		
418	tagptr -> lase_intr_enable = 0;		
419	tagptr -> lase_intr_enable = 1;		
420	/*		
421	no one is driving error.		
422	*/		
423	found_source_of_interrupt=1;		
424	}		
425	if( fatal == 1 )		
426	{		
427	/*		
428	found source of error, but it was fatal.		
429	*/		
430	found_source_of_interrupt=1;		
431	if( modeler_error.tag_error == 1 &&		
432	(( lases_enable & lases_short ) && multi_lases > 1) )		
433	modeler_error.tag_error = 0;		
434	else		
435	{		
436	/*		
437	save in NVRam		
438	*/		
439	(void)lm_nvrwram_access((char *) &modeler_error,		
440	HARDWARE_ERROR, sizeof HARDWARE_ERROR,		
441	MEMORY_WRITE, &error);		
442	/*		
443	reset cpu(BACKPLANE_ERROR);		
444	}		
445	}		
446	if( found_source_of_interrupt == 1 )		
447	{		
448	unknown_source_of_interrupt = 0;		
449	}		
450	else		
451	{		
452	if( unknown_source_of_interrupt++ > 2 )		
453	{		
454	(void)disable_mod_err();		
455	printf("Unknown : rce of backplane interrupt\n");		
456	modeler_error.unknown_source_of_interrupt=1;		
457	}		
458	}		
459	}		
460	}		
461	void		
462	enable_mod_err()		
463	{		
464	cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;		
465	/*		
466	printf("enable_mod_err()\n");		
467	** enable clock board interrupts		
468	*/		
469	p_cpu_ctl_reg->tag_intr_ena = 1;		
470	}		
471	void		
472	disable_mod_err()		
473	{		
474	cpu_control_reg_struct *p_cpu_ctl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;		
475	/*		
476	printf("disable_mod_err()\n");		
477	** enable clock board interrupts		
478	*/		
479	}		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/mod_err.c	DATE 5/23/89 TIME 4:42:15 pm	PAGE # 5/48
LINE #	SOURCE TEXT			
480	/*			
481	P_cpu_ctl_reg->img_intr_err = 0;			
482	*/			
483	reset_cpu_os_fatal_error()			
484	{			
485	unsigned long error;			
486	/*			
487	** save in NVRAM			
488	*/			
489	(void)lm_nvram_access((char *) &modeler_error, HARDWARE_ERROR, sizeof(HARDWARE_ERROR), MEMORY_WRITE, &error);			
490	reset_cpu(BACKPLANE_ERROR);			
491	}			
492	/*			
493	*/			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/nvsram.c

DATE \* 5/23/89  
TIME 4:42:16 pm

PAGE #  
1/49

```

1  // SCCS_ID: nvsram.c Rev 3.1, 4/24/89 at 07:47:12
2  /*
3  ** Routines to access NVSRAM
4  */
5  #include "common.h"
6  #include "cpu.h"
7  #include "ls_rd_wr.h"
8  #include "mod_err.h"
9  #include "nvsram.h"
10 #include "uart.h"
11
12 char modular_state = 0; /* undefined */
13
14 static char sram_init = 0;
15 char ls_valid_nvsram = FALSE;
16
17 /*
18 ** Compute checksum of NVSRAM, passed pointer to start, returns as u_long
19 */
20 u_long
21 nvsram_compute_checksum(sram)
22 u_long *sram;
23 {
24     register u_short i;
25     register u_long checksum;
26
27     checksum = 0;
28     for(i = 0; i < (CPU_NVSRAM_SIZE / 4); i++)
29     {
30         checksum += (*sram + 0xffffffff);
31         sram++;
32     }
33     /*
34     ** Return 2's complement
35     */
36     return (((~checksum) + 0xffffffff) + 0x01000000);
37 }
38
39 u_short
40 ls_nvsram_access(data_struct, field, sizeof_field, option, error)
41 register Char *data_struct;
42 register u_long field;
43 register u_long sizeof_field, option;
44 register u_long *error;
45 {
46     u_short version;
47     register u_long *sram = (u_long *) (CPU_NVSRAM + field);
48     cpu_control_reg_struct *p_cpu_ctrl_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;
49     register u_long checksum;
50     register u_short i;
51     *error = NO_NVSRAM_ERROR;
52
53     /*
54     ** If reading or writing we have valid sram
55     ** and sram access is within bounds
56     */
57     if(option == MEMORY_READ || option == MEMORY_WRITE)
58     {
59         if((field + sizeof_field) > CPU_NVSRAM_SIZE)
60         {
61             *error = NO_NVSRAM;
62             return (FAILURE);
63         }
64         if(!sram_init)
65         {
66             *error = INVALID_NVSRAM;
67             return (FAILURE);
68         }
69     }
70
71     switch (option)
72     {
73     case MEMORY_READ:
74         /*
75         ** perform read
76         */
77         for(i = 0; i < sizeof_field; i++)
78         {
79             *data_struct++ = (u_char) (*sram >> 24);
80             sram++;
81         }
82         return (SUCCESS);
83     case MEMORY_WRITE:
84         /*
85         ** perform write
86         */
87         p_cpu_ctrl_reg->nvsram_write_ena = 1;
88         for(i = 0; i < sizeof_field; i++)
89         {
90             *sram++ = (u_long) *data_struct << 24;
91             data_struct++;
92         }
93         /*
94         ** setup pointer again
95         */
96         sram = (u_long *) (CPU_NVSRAM + CHECKSUM);
97         /*
98         ** clear previous checksum, calculate, and store
99         */
100        *sram = 0;
101        checksum = nvsram_compute_checksum(sram);
102        *sram = checksum;
103        p_cpu_ctrl_reg->nvsram_write_ena = 0;
104        return (SUCCESS);
105    case MEMORY_INIT:
106        /*
107        ** Initialize sram
108        */
109        p_cpu_ctrl_reg->nvsram_write_ena = 1;
110        /*
111        ** write signature
112        */
113        sram = (u_long *) (CPU_NVSRAM + BSIC);
114        *sram++ = (u_long) 'l' << 24;
115        *sram++ = (u_long) 'm' << 24;
116        *sram++ = (u_long) 's' << 24;
117        *sram++ = (u_long) 'i' << 24;
118        /*
119        ** initialize rest of sram to 0
120        */
121    }
122 }

```

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/nvsram.c

DATE	5/23/89	PAGE #
TIME	4:42:16 pm	2/50

LINE #	SOURCE TEXT
121	/*
122	for(i = 4; i < (CPU_NVSRAM_SIZE / 4); i++)
123	*aram++ = 0;
124	/*
125	set up BAUD rate
126	*/
127	aram = (u_long *) CPU_NVSRAM;
128	*(u_long *) ((int) aram + BAUD_RATEA) =
129	(u_long) (BAUD_96_12_24 << 24);
130	*(u_long *) ((int) aram + BAUD_RATEB) =
131	(u_long) (BAUD_24_96_12 << 24);
132	/*
133	compute and store checksum
134	*/
135	checksum = nvsram_compute_checksum(aram);
136	*(u_long *) ((int) aram + CHECKSUM) = checksum;
137	*/
138	aram_init = 1;
139	p_cpu_ctrl_ewy->nvsram_write_err = 0;
140	return (SUCCESS);
141	case MEMORY_VALIDATE:
142	/*
143	validate NVSRAM signature
144	*/
145	aram = (u_long *) (CPU_NVSRAM + BSIG);
146	/*
147	check for aram initialized, but without a boot
148	structure
149	*/
150	if(((aram >> 24) != 'L')
151	((aram >> 24) != 'M')
152	((aram >> 24) != 'S')
153	((aram >> 24) != 'I'))
154	{
155	error = INVALID_NVSRAM;
156	return (FAILURE);
157	}
158	/*
159	calculate checksum and verify
160	*/
161	checksum = nvsram_compute_checksum((u_long *) CPU_NVSRAM);
162	if (checksum != 0)
163	{
164	error = INVALID_NVSRAM;
165	return (FAILURE);
166	}
167	/*
168	validate NVSRAM version
169	*/
170	version = (u_short) *(u_char *) (CPU_NVSRAM + BVER + 1*4);
171	if (version < EPROM_NVSRAM_VERSION) {
172	error = STALE_HOST_SOFTWARE;
173	return (FAILURE);
174	}
175	else if (version > EPROM_NVSRAM_VERSION) {
176	error = STALE_EPROM_SOFTWARE;
177	return (FAILURE);
178	}
179	in_valid_nvsram = TRUE;
180	return (SUCCESS);
181	case MEMORY_DIAG_INT:
182	aram_init = 1;
183	return (SUCCESS);
184	default:
185	return (FAILURE);
186	}
187	}
188	}
189	}
190	}

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/parity.c	DATE 5/23/89	PAGE # 1/51
LINE #		SOURCE TEXT		
1		/* SCCS ID: parity.c rev 1.1, 4/24/89 at 07:47:16 */		
2		/*		
3		** Parity error handler		
4		**		
5		*/		
6		#include "common.h"		
7		#include "cpu.h"		
8		#include "mod_err.h"		
9		#include "svarm.h"		
10		#include "lm_rd_wr.h"		
11		extern u_short lm_svarm_access();		
12		extern void output_routine(), reset_cpu();		
13		parity_error()		
14		{		
15		register cpu_control_reg_struct *control_reg = (cpu_control_reg_struct *) CPU_CONTROL_REG;		
16		register cpu_par_err_reg *parity_reg = (cpu_par_err_reg *) CPU_PAR_ERR_REG, parity;		
17		u_long error;		
18		char buffer[ 100 ];		
19		{		
20		parity = *parity_reg;		
21		if (control_reg->not_parity_intr == 0)		
22		{		
23		sprintf(buffer, "Parity error addr = %08x\n",		
24		parity_reg->error_addr * 4);		
25		output_routine(buffer);		
26		sprintf(buffer, "68020 to VME to LANCE to\n",		
27		parity_reg->not_68020_master,		
28		parity_reg->not_VME_master,		
29		parity_reg->not_LANCE_master);		
30		output_routine(buffer);		
31		sprintf(buffer, "hi to um to lm to lo to\n",		
32		parity_reg->not_error_hi,		
33		parity_reg->not_error_um,		
34		parity_reg->not_error_lm,		
35		parity_reg->not_error_lo);		
36		output_routine(buffer);		
37		(void)lm_svarm_access((char *) parity_reg, PARITY_ERR, sizeof(PARITY_ERR), MEMORY_WRITE, &error);		
38		control_reg->parity_force = 0;		
39		reset_cpu(PARITY_ERROR);		
40		}		
41		}		
42		}		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/timer.c	DATE 5/23/89 TIME 4:42:16 pm	PAGE # 1/52
SOURCE TEXT				
LINE #				
1	/* SCCS ID: timer.c Rev 3.1, 4/24/89 at 07:47:13 */			
2	/*			
3	** Initialize all three timers.			
4	*/			
5	#include "common.h"			
6	#include "cpu.h"			
7	#include "timer.h"			
8				
9				
10	init_all_timer()			
11	{			
12	unsigned long *timer;			
13	timer_control = timer_control_reg;			
14	timer_control timer_254_control;			
15				
16	/*			
17	** set up timer 0			
18	*/			
19	/* write the control register			
20	** timer 0, write lsb then msb, use binary value for counter.			
21	*/			
22	timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER0;			
23	timer_254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR;			
24	timer_254_control.timer_mode = TIMER_MODE3;			
25	timer_254_control.bcd = BINARY;			
26	/*			
27	** set up a pointer to the 254 timer chip, control register			
28	** and setup timer 0			
29	*/			
30	timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTRL_WORD);			
31	timer_control_reg = timer_254_control;			
32	/*			
33	** write LSB then MSB counter regs to obtain 5 ms waveform			
34	** ( 0x1400 ) = 1024000 / 200;			
35	*/			
36	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);			
37	*timer = 0x00000000;			
38	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);			
39	*timer = 0x14000000;			
40	/*			
41	** set up timer 1			
42	*/			
43	/*			
44	** write the control register			
45	** timer 1, write lsb then msb, use binary value for counter.			
46	*/			
47	timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER1;			
48	/*			
49	** write to the 254 timer chip, control register			
50	*/			
51	timer_control_reg = timer_254_control;			
52	/*			
53	** write LSB then MSB counter regs to obtain 5 ms waveform			
54	** 320 ( 0x140 ) = 64000 / 200;			
55	*/			
56	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);			
57	*timer = 0x40000000;			
58	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);			
59	*timer = 0x01000000;			
60	/*			
61	** set up timer 2			
62	*/			
63	/*			
64	** write the control register			
65	** set up timer mode 0			
66	** This provides tick for a/v clock			
67	** timer 2, write lsb then msb, use binary value for counter.			
68	*/			
69	timer_254_control.timer_mode = TIMER_MODE0;			
70	timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER2;			
71	/*			
72	** write to the 254 timer chip, control register			
73	*/			
74	timer_control_reg = timer_254_control;			
75	/*			
76	** write LSB then MSB counter regs to obtain 5 ms waveform			
77	** 320 ( 0x140 ) = 64000 / 200;			
78	*/			
79	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER2);			
80	*timer = 0x40000000;			
81	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER2);			
82	*timer = 0x01000000;			
83	/*			
84	**			
85	/*			
86	** setup timer 0 to trigger after time microseconds			
87	*/			
88	static u_short time_to_wait;			
89	void			
90	setup_timer0( time )			
91	unsigned long time;			
92	{			
93	unsigned long *timer;			
94	timer_control = timer_control_reg;			
95	timer_control timer_254_control;			
96				
97	/*			
98	** set up timer 0			
99	*/			
100	time_to_wait = time - ((1024000 / ( 1000000 / time )) & 0xffff)-1;			
101	/*			
102	** write the control register			
103	** timer 0, write lsb then msb, use binary value for counter.			
104	*/			
105	timer_254_control.select_timer_counter = SELECT_TIMER_COUNTER0;			
106	timer_254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR;			
107	timer_254_control.timer_mode = TIMER_MODE0;			
108	timer_254_control.bcd = BINARY;			
109	/*			
110	** set up a pointer to the 254 timer chip, control register			
111	** and setup timer 0			
112	*/			
113	timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTRL_WORD);			
114	timer_control_reg = timer_254_control;			
115	/*			
116	** write LSB then MSB counter regs			
117	*/			
118	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);			
119	*timer = time << 24;			
120	timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);			

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/timer.c	DATE * 5/23/89	PAGE # 2/53
LINE #		SOURCE TEXT		
121		*timer = time << 16;		
122		}		
123		/*		
124		** returns FAILURE if timer0 has not expired		
125		** else SUCCESS		
126		** Must call setup_timer0( microseconds_to_time );		
127		** before checking.		
128		**		
129		/*		
130		unsigned short		
131		read_timer0(elapsed);		
132		u_long *elapsed;		
133		{		
134		counter_status timer_0_status, *ptr_timer_0_status = (counter_status *) (CPU_TIMER + TIMER_COUNTER0);		
135		timer_status *timer_status_reg;		
136		timer_status timer_8254_status;		
137		unsigned long *timer;		
138		u_short time;		
139		{		
140		/*		
141		** read timer 0 status		
142		** The bits are not the same		
143		*/		
144		timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */		
145		timer_8254_status.count_status = READ_BACK_STATUS; /* read status */		
146		timer_8254_status.counter = READ_TIMER0; /* timer 0 */		
147		timer_8254_status.zero = 0; /* must be 0 */		
148		/*		
149		** set up a pointer to the 8254 timer chip, control register		
150		** and setup timer 0		
151		/*		
152		timer_status_reg = (timer_status *) (CPU_TIMER + TIMER_CNTL_WORD);		
153		*timer_status_reg = timer_8254_status;		
154		timer_0_status = *ptr_timer_0_status;		
155		{		
156		if( timer_0_status.output == OUT_HIGH ) {		
157		return( SUCCESS );		
158		}		
159		else		
160		{		
161		/*		
162		** read timer 0 count		
163		*/		
164		timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */		
165		timer_8254_status.count_status = READ_BACK_COUNT; /* read count */		
166		timer_8254_status.counter = READ_TIMER0; /* timer 0 */		
167		timer_8254_status.zero = 0; /* must be 0 */		
168		*timer_status_reg = timer_8254_status;		
169		{		
170		timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER0);		
171		{		
172		time = (u_short)((*timer >> 24) & 0xff);		
173		time  = (u_short)((*timer >> 16) & 0xff00);		
174		time = time_to_wait - time;		
175		*elapsed = (u_long)time * 1000 / 1024;		
176		return( FAILURE );		
177		}		
178		}		
179		/*		
180		** setup timer 1 to trigger after time microseconds		
181		*/		
182		void		
183		setup_timer1( time )		
184		{		
185		unsigned long time;		
186		{		
187		unsigned long *timer;		
188		timer_control *timer_control_reg;		
189		timer_control timer_8254_control;		
190		{		
191		/*		
192		** set up timer 1		
193		*/		
194		time = ((64000 / (1000000 / time)) & 0xffff) - 1;		
195		/*		
196		** write the control register		
197		** timer 1, write lsb then msb, use binary value for counter.		
198		*/		
199		timer_8254_control.select_timer_counter = SELECT_TIMER_COUNTER1;		
200		timer_8254_control.r_w_timer_counter = R_W_LSB_MSB_TIMER_CNTR;		
201		timer_8254_control.timer_mode = TIMER_MODE0;		
202		timer_8254_control.bcd = BINARY;		
203		/*		
204		** set up a pointer to the 8254 timer chip, control register		
205		** and setup timer 1		
206		/*		
207		timer_control_reg = (timer_control *) (CPU_TIMER + TIMER_CNTL_WORD);		
208		*timer_control_reg = timer_8254_control;		
209		{		
210		** write lsb then msb counter regs		
211		/*		
212		timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);		
213		*timer = time << 24;		
214		*timer = (unsigned long *) (CPU_TIMER + TIMER_COUNTER1);		
215		*timer = time << 16;		
216		}		
217		/*		
218		** returns FAILURE if timer0 has not expired		
219		** else SUCCESS		
220		** Must call setup_timer1( microseconds_to_time );		
221		** before checking.		
222		**		
223		/*		
224		unsigned short		
225		read_timer1( )		
226		{		
227		counter_status timer_1_status, *ptr_timer_1_status = (counter_status *) (CPU_TIMER + TIMER_COUNTER1);		
228		timer_status *timer_status_reg;		
229		timer_status timer_8254_status;		
230		{		
231		/*		
232		** read timer 1 status		
233		** The bits are not the same		
234		*/		
235		timer_8254_status.read_back = READ_BACK_TIMER; /* READ BACK COMMAND */		
236		timer_8254_status.count_status = READ_BACK_STATUS; /* read status */		
237		timer_8254_status.counter = READ_TIMER1; /* timer 1 */		
238		timer_8254_status.zero = 0; /* must be 0 */		
239		/*		
240		** set up a pointer to the 8254 timer chip, control register		



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM os/timer.c	DATE > 5/23/89 TIME 4:42:16 pm	PAGE # 3/54
LINE #	SOURCE TEXT			
241	** and setup timer 1			
242	*/			
243	timer_status_reg = (timer_status *) (CPU_TIMER + TIMER_CNTL_WORD);			
244	*timer_status_reg = timer_0254_status;			
245				
246	timer_1_status = *ptr_timer_1_status;			
247				
248	if( timer_1_status.output == OUT_HIGH )			
249	return( SUCCESS );			
250	else			
251	{			
252	return( FAILURE );			
253	}			
254	}			

Copyright 1989  
Logic Modeling Systems

SOURCE PROGRAM  
os/vrtxkeys.c

#	DATE	5/23/89	PAGE #
	TIME	4:42:16 pm	1/55

```

1  // SCCS ID: vrtxkeys.c rev 1.1, 4/24/89 at 07:47:22
2
3  int key_hit;
4
5  #define GET_KEY 1
6  #define SIZEOFARRAY 0x1f /* MUST ALWAYS BE A POWER OF 2 minus 1 */
7  static char in_ptr = 0, out_ptr = 0, count_keys = 0, array[SIZEOFARRAY+1],
8
9  /*
10  -- buffer up keys, as they arrive
11  -- this is a task and it should be given the highest priority
12  -- count_keys keeps track of keys.
13  */
14  void
15  put_getc()
16  {
17  char ch;
18  long err;
19  while( 1 )
20  {
21      ch = sc_getc();
22      if( count_keys < SIZEOFARRAY )
23      {
24          array[ in_ptr++ ] = ch;
25          in_ptr %= SIZEOFARRAY;
26          count_keys++;
27          count_keys %= SIZEOFARRAY;
28          sc_send( key_hit, &err );
29          if( err )
30              printf( "error posting to semaphore" );
31      }
32  }
33
34  /*
35  -- stick around till a key is entered
36  */
37  get_key()
38  {
39  int err;
40  char ch;
41  while( 1 )
42  {
43      sc_send( key_hit, 0, &err );
44      if( err )
45          printf( "error pending for semaphore" );
46      if( count_keys )
47      {
48          count_keys--;
49          ch = array[ out_ptr++ ];
50          out_ptr %= SIZEOFARRAY;
51          return( ch );
52      }
53  }
54
55  }
56
57  /*
58  -- return number of unprocessed keys
59  */
60  check_key()
61  {
62      return( count_keys );
63  }
64
65  }

```

SOURCE PROGRAM		DATE	PAGE #
misc/message.c		5/23/89	1/1
TIME		1:20:42 pm	
SOURCE TEXT			
LINE #			
1	/* SCCS ID: message.c; 4/24/89 at 07:17:13 */		
2	#ifndef MODEL		
3	/* On the host, use the definition for... */		
4	#include <stdio.h>		
5	#endif		
6	#include <errno.h>		
7	#include <varargs.h>		
8	/* On the modular, use the definition for sprintf() */		
9	#include "common.h"		
10	#include "message.h"		
11	#endif VMS		
12	/* descrip */		
13	#include		
14	#endif		
15	#define ABSOLUTE_MAXIMUM_MESSAGE 1024		
16	typedef struct message {		
17	u_short message_severity;		
18	char message_text;		
19	struct message *next_message;		
20	} MESSAGE;		
21	static u_short error_message_present = FALSE;		
22	static u_short warning_message_present = FALSE;		
23	static MESSAGE *start_of_message_list = (MESSAGE *) NULL;		
24	static MESSAGE *end_of_message_list = (MESSAGE *) NULL;		
25	static MESSAGE *out_of_memory;		
26	static u_short no_memory_already_linked = FALSE;		
27	static char *no_memory_message = "cannot allocate memory to store error/warning message";		
28	static		
29	error_count = 0;		
30	static		
31	warning_count = 0;		
32	extern void link_message();		
33	extern void queue_message();		
34	extern void get_system_string();		
35	extern void handle_out_of_memory();		
36			
37			
38			
39			
40			
41			
42			
43			
44	/*VARIABLES*/		
45	void		
46	link_message(va_list)		
47	{		
48	{		
49	va_list args;		
50	u_long type;		
51	char *format;		
52	int error_number;		
53	char system_message[SYS_MESSAGE];		
54	char ln_msg[ABSOLUTE_MAXIMUM_MESSAGE];		
55			
56	va_start(args);		
57	type = (u_long) va_arg(args, u_long);		
58	if (type == VMS_ERROR_MSG    type == VMS_WARNING_MSG)		
59	error_number = va_arg(args, int);		
60	else if (type == SYS_ERROR_MSG    type == SYS_WARNING_MSG)		
61	error_number = errno;		
62	format = va_arg(args, char *);		
63	(void) vsprintf(ln_msg, format, args);		
64	va_end(args);		
65	if (type == SYS_ERROR_MSG    type == SYS_WARNING_MSG		
66	type == VMS_ERROR_MSG    type == VMS_WARNING_MSG		
67	{		
68	get_system_string(error_number, system_message);		
69	(void) strcat(ln_msg, system_message);		
70	}		
71	if (type == ERROR_MSG    type == SYS_ERROR_MSG    type == VMS_ERROR_MSG) {		
72	++error_count; /* does not work if out of memory */		
73	error_message_present = TRUE;		
74	queue_message(ERROR_MSG, ln_msg);		
75	}		
76	else {		
77	++warning_count; /* does not work if out of memory */		
78	warning_message_present = TRUE;		
79	queue_message(WARNING_MSG, ln_msg);		
80	}		
81	}		
82			
83			
84			
85			
86			
87			
88			
89			
90			
91	void		
92	link_message_queue()		
93	{		
94	register MESSAGE *message;		
95			
96			
97	error_count = 0;		
98	warning_count = 0;		
99	error_message_present = FALSE;		
100	warning_message_present = FALSE;		
101	while(start_of_message_list != (MESSAGE *) NULL) {		
102	message = start_of_message_list;		
103	start_of_message_list = start_of_message_list->next_message;		
104			
105			
106			
107	if (message == out_of_memory) {		
108	no_memory_already_linked = FALSE;		
109	}		
110	else {		
111	(void) free(message->message_text);		
112	(void) free((char *)message);		
113	}		
114	}		
115			
116			
117	u_short		
118	link_message_queue(type, str)		
119	u_short *type;		
120	char *str;		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM misc/message.c	DATE 5/23/89	PAGE # 2/2
LINE #		SOURCE TEXT		
21		{		
22		register MESSAGE *old_message;		
23		if (start_of_message_list == (MESSAGE *) NULL) {		
24		error_message_present = FALSE;		
25		warning_message_present = FALSE;		
26		error_count = 0;		
27		warning_count = 0;		
28		return(FAILURE);		
29		}		
30		*type = start_of_message_list->message_severity;		
31		(void) strcpy(str, start_of_message_list->message_text);		
32		if (*type == ERROR_MSG)		
33		error_count++;		
34		else		
35		warning_count++;		
36		old_message = start_of_message_list;		
37		start_of_message_list = start_of_message_list->next_message;		
38		if (old_message != test_of_memory) {		
39		(void) free(old_message->message_text);		
40		(void) free((char *)old_message);		
41		}		
42		else {		
43		no_memory_already_linked = FALSE;		
44		}		
45		return(SUCCESS);		
46		}		
47		u_short		
48		is_remove_message(type, str)		
49		u_short type;		
50		char *str;		
51		{		
52		register MESSAGE *message;		
53		register MESSAGE *old_message;		
54		if (start_of_message_list == (MESSAGE *) NULL) {		
55		return(FAILURE);		
56		}		
57		message = start_of_message_list;		
58		old_message = start_of_message_list;		
59		while(message != (MESSAGE *) NULL) {		
60		if ( (type == message->message_severity) &&		
61		(strcmp(message->message_text, str) == 0) )		
62		{		
63		/* Verify the message we are trying to get rid of */		
64		old_message->next_message = message->next_message;		
65		if (old_message == message)		
66		start_of_message_list = old_message->next_message;		
67		if (message != test_of_memory) {		
68		(void) free(message->message_text);		
69		(void) free((char *)message);		
70		}		
71		if (--error_count == 0)		
72		error_message_present = FALSE;		
73		return(SUCCESS);		
74		}		
75		old_message = message;		
76		message = message->next_message;		
77		}		
78		return(FAILURE);		
79		}		
80		void		
81		is_message_types(errors, warnings)		
82		u_short *errors;		
83		u_short *warnings;		
84		{		
85		*errors = error_count;		
86		*warnings = warning_count;		
87		}		
88		static void		
89		get_system_string(error_number, str)		
90		int		
91		error_number;		
92		char *str;		
93		{		
94		#ifdef VMS		
95		extern int sys_err;		
96		extern char *sys_errlist[];		
97		if (error_number < 0)		
98		(void) sprintf(str, "");		
99		else if (error_number < sys_err)		
100		(void) sprintf(str, "%s", sys_errlist[error_number], error_number);		
101		else (void) sprintf(str, "%s", error_number);		
102		}		
103		#else /* VMS */		
104		{		
105		unsigned short message_length;		
106		unsigned long status;		
107		unsigned long flags = 0x0F;		
108		struct dcdsdescriptor message_descriptor;		
109		struct {		
110		unsigned char reserved1;		
111		unsigned char fce_count;		
112		unsigned char user_value;		
113		unsigned char reserved2;		
114		} status_struct;		
115		message_descriptor.dcds_class = DCDSE_CLASS_S;		
116		message_descriptor.dcds_dtype = DCDSE_DTYPE_T;		
117		message_descriptor.dcds_length = 256; /* max VMS message */		
118		message_descriptor.dcds_pointer = str;		
119		status = SYSGETDCDS(error_number, message_length,		
120		message_descriptor, flags, &status_struct);		

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM	DATE	PAGE #
	misc/message.c	5/23/89	3/3
		TIME	1:20:42 pm

LINE #	SOURCE TEXT
241	if (! (status & 1))
242	LIBSIGNAL (status);
243	
244	str[message_length] = '\0'; /* null terminate string */
245	
246	endif /* VMS */
247	
248	
249	
250	
251	static void
252	queue_message(severity, str)
253	u_short severity;
254	char *str;
255	{
256	register int length;
257	register MESSAGE *message;
258	
259	message = (MESSAGE *) malloc((unsigned) sizeof(MESSAGE));
260	if (message == (MESSAGE *) NULL) {
261	handle_out_of_memory();
262	return;
263	}
264	message->message_severity = severity;
265	
266	if ((length-strlen(str)) >= 256) {
267	length = 256;
268	str[length] = '\0';
269	}
270	message->message_text = malloc((unsigned)(length+1));
271	if (message->message_text == (char *) NULL) {
272	free((char *) message);
273	handle_out_of_memory();
274	return;
275	}
276	(void) strcpy(message->message_text, str);
277	
278	link_message(message);
279	
280	
281	
282	
283	static void
284	handle_out_of_memory()
285	{
286	if (no_memory_already_linked == FALSE) {
287	no_memory_already_linked = TRUE;
288	out_of_memory.message_severity = ERROR_MSG;
289	out_of_memory.message_text = no_memory_message;
290	link_message(&out_of_memory);
291	}
292	
293	
294	static void
295	link_message(message)
296	MESSAGE *message;
297	{
298	if (start_of_message_list == (MESSAGE *) NULL) {
299	start_of_message_list = end_of_message_list = message;
300	}
301	else {
302	end_of_message_list->next_message = message;
303	end_of_message_list = message;
304	}
305	
306	end_of_message_list->next_message = (MESSAGE *) NULL;
307	}

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM misc/resolve.c	DATE 5/23/89 TIME 11:20:42 pm	PAGE # 1/4
LINE #	SOURCE TEXT		
1	/* SCCS ID: resolve.c; rev. 3.4; 5/9/89; at: 15:44:24 */		
2	#ifndef VMS		
3	#include <sys/types.h>		
4	#include <sys/stat.h>		
5	#ifdef SYS5		
6	#include <direct.h>		
7	#else		
8	#include <sys/dir.h>		
9	#endif		
10	#include <stdio.h>		
11	#include <ctype.h>		
12	#include "lm_sfi.h"		
13	#include "common.h"		
14	#include "message.h"		
15	#define max_str 256		
16	long check_all_sub_dirs();		
17	char *getenv();		
18	lm_check_path_variable (lm_path)		
19	char *lm_path;		
20	{		
21	char *p, *pos, dir[max_str], pathname[1024];		
22	char *end_of_string;		
23	long accessible;		
24	char delimiter[max_str];		
25	long status;		
26	struct stat dir_stat;		
27	/* get the path to search */		
28	p = (char *) getenv (lm_path);		
29	if (NULL == p) {		
30	lm_message(ERROR_MSG, "environment variable \"%s\" is not defined ",		
31	lm_path);		
32	return (LM_ERROR);		
33	}		
34	if ((strlen (p) > sizeof (pathname))) {		
35	lm_message(ERROR_MSG, "environment variable \"%s\" is too long ", lm_path);		
36	return (LM_ERROR);		
37	}		
38	(void) strcpy (pathname, p);		
39	p = pathname;		
40	end_of_string = pathname + strlen(pathname);		
41	/* find out if it's a colon or space separated list */		
42	pos = (char *) strchr (pathname, ":");		
43	if (NULL == pos) pos = pathname + strlen(pathname);		
44	(void) strcpy (delimiter, pos, 1);		
45	delimiter[1] = NULL;		
46	/* look for the file in each element of the path */		
47	status = LM_SUCCESS;		
48	do {		
49	pos = (char *) strchr (p, delimiter);		
50	if (NULL == pos) pos = pathname + strlen(pathname);		
51	(void) strcpy (dir, p, pos-p);		
52	dir[pos-p] = NULL;		
53	p = pos + 1;		
54	accessible = lm_access(dir, 0);		
55	if (accessible != 0) {		
56	lm_message(WARNING_MSG, "directory \"%s\" is \"%s\" does not exist or cannot be accessed ", dir, lm_path);		
57	status = LM_WARNING;		
58	} else {		
59	stat (dir, &dir_stat);		
60	if (!(dir_stat.st_mode & S_IFDIR)) {		
61	lm_message(WARNING_MSG, "\"%s\" is \"%s\" is not a directory ", dir, lm_path);		
62	status = LM_WARNING;		
63	}		
64	} while (pos != end_of_string);		
65	return (status);		
66	}		
67	static int		
68	lm_access (filename, read_flag)		
69	char *filename;		
70	int read_flag;		
71	{		
72	#ifdef SYS_7		
73	#define shell_var "SHELL"		
74	#define cegis_shell "/csm/sh"		
75	/* begin filename resolution. The file is assumed to be either		
76	/* (1) exactly as the user entered it (2) exactly as the user		
77	/* entered it, but with .x instead of .; (3) entirely uppercase		
78	/* on the disk (as distributed by MUI) or (4) entirely lowercase		
79	/* on the disk (default for files created under cegis 3.7)		
80	/*		
81	/* The user may completely ignore case sensitivity problems with		
82	/* the following exceptions:		
83	/*		
84	/* 1) The file is mixed case on the disk		
85	/* 2) The file is on a disk NFSed to a true Unix machine		
86	/*		
87	/* In these cases, the filename must be entered as it resides on		
88	/* the disk.		
89	if (getenv(shell_var) && (0 == strcmp(cegis_shell, getenv(shell_var)))) {		
90	120		

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM misc/resolve.c	DATE 5/23/89	PAGE # 2/5
--	--	----------------------------------	-----------------	---------------

LINE #	SOURCE TEXT
121	char local_filename[max_str]; /* Don't change the filename in here. */
122	char old_local_filename[max_str];
123	char *filename_pointer;
124	char *old_filename;
125	/* Check the filename as entered by the user. If we find
126	/* it great, if not, keep hunting */
127	if (0 == access(filename, read_flag))
128	return(0);
129	/* convert all uppercase to lower, 'x' to 'X' for all
130	/* characters including the path. */
131	strcpy(local_filename, filename);
132	filename_pointer = local_filename;
133	old_filename = filename;
134	while (*old_filename) {
135	if (isupper(*old_filename)) {
136	*filename_pointer++ = 'x';
137	*filename_pointer++ = tolower(*old_filename++);
138	} else
139	*filename_pointer++ = *old_filename++;
140	}
141	*filename_pointer = NULL;
142	if (0 == access(local_filename, read_flag)) {
143	strcpy(filename, local_filename);
144	return(0);
145	}
146	/* remove all '/'s (except '/'s in the filename, NOT
147	/* the path, and try again. */
148	filename_pointer = local_filename;
149	if (strchr(filename_pointer, '/'))
150	filename_pointer = strchr(filename_pointer, '/');
151	while (*filename_pointer) {
152	if ((*filename_pointer == '/') && ((filename_pointer+1) != '\0'))
153	strcpy(filename_pointer, filename_pointer+1);
154	filename_pointer++;
155	}
156	*filename_pointer = NULL;
157	if (0 == access(local_filename, read_flag)) {
158	strcpy(filename, local_filename);
159	return(0);
160	}
161	/* add '/'s to all characters (assume all CAPS) in the
162	/* filename NOT the path, and try yet again. */
163	strcpy(old_local_filename, local_filename);
164	filename_pointer = local_filename;
165	if (strchr(filename_pointer, '/'))
166	filename_pointer = strchr(filename_pointer, '/');
167	old_filename = old_local_filename;
168	if (strchr(old_filename, '/'))
169	old_filename = strchr(old_filename, '/');
170	while (*old_filename) {
171	if (isalpha(*old_filename)) {
172	*filename_pointer++ = 'X';
173	}
174	*filename_pointer++ = *old_filename++;
175	}
176	*filename_pointer = NULL;
177	if (0 == access(local_filename, read_flag)) {
178	strcpy(filename, local_filename);
179	return(0);
180	}
181	}
182	/* If not running single I/O, just return the access() */
183	return (access(filename, read_flag));
184	}
185	/* If not running single I/O, just return the access() */
186	return (access(filename, read_flag));
187	}
188	/* If not running single I/O, just return the access() */
189	return (access(filename, read_flag));
190	}
191	/* If not running single I/O, just return the access() */
192	return (access(filename, read_flag));
193	}
194	/* If not running single I/O, just return the access() */
195	return (access(filename, read_flag));
196	}
197	/* If not running single I/O, just return the access() */
198	return (access(filename, read_flag));
199	}
200	/* If not running single I/O, just return the access() */
201	return (access(filename, read_flag));
202	}
203	/* If not running single I/O, just return the access() */
204	return (access(filename, read_flag));
205	}
206	/* If not running single I/O, just return the access() */
207	return (access(filename, read_flag));
208	}
209	/* If not running single I/O, just return the access() */
210	return (access(filename, read_flag));
211	}
212	/* If not running single I/O, just return the access() */
213	return (access(filename, read_flag));
214	}
215	/* If not running single I/O, just return the access() */
216	return (access(filename, read_flag));
217	}
218	/* If not running single I/O, just return the access() */
219	return (access(filename, read_flag));
220	}
221	/* If not running single I/O, just return the access() */
222	return (access(filename, read_flag));
223	}
224	/* If not running single I/O, just return the access() */
225	return (access(filename, read_flag));
226	}
227	/* If not running single I/O, just return the access() */
228	return (access(filename, read_flag));
229	}
230	/* If not running single I/O, just return the access() */
231	return (access(filename, read_flag));
232	}
233	/* If not running single I/O, just return the access() */
234	return (access(filename, read_flag));
235	}
236	/* If not running single I/O, just return the access() */
237	return (access(filename, read_flag));
238	}
239	/* If not running single I/O, just return the access() */
240	return (access(filename, read_flag));
241	}
242	/* If not running single I/O, just return the access() */
243	return (access(filename, read_flag));
244	}
245	/* If not running single I/O, just return the access() */
246	return (access(filename, read_flag));
247	}
248	/* If not running single I/O, just return the access() */
249	return (access(filename, read_flag));
250	}
251	/* If not running single I/O, just return the access() */
252	return (access(filename, read_flag));
253	}
254	/* If not running single I/O, just return the access() */
255	return (access(filename, read_flag));
256	}
257	/* If not running single I/O, just return the access() */
258	return (access(filename, read_flag));
259	}
260	/* If not running single I/O, just return the access() */
261	return (access(filename, read_flag));
262	}
263	/* If not running single I/O, just return the access() */
264	return (access(filename, read_flag));
265	}
266	/* If not running single I/O, just return the access() */
267	return (access(filename, read_flag));
268	}
269	/* If not running single I/O, just return the access() */
270	return (access(filename, read_flag));
271	}
272	/* If not running single I/O, just return the access() */
273	return (access(filename, read_flag));
274	}
275	/* If not running single I/O, just return the access() */
276	return (access(filename, read_flag));
277	}
278	/* If not running single I/O, just return the access() */
279	return (access(filename, read_flag));
280	}
281	/* If not running single I/O, just return the access() */
282	return (access(filename, read_flag));
283	}
284	/* If not running single I/O, just return the access() */
285	return (access(filename, read_flag));
286	}
287	/* If not running single I/O, just return the access() */
288	return (access(filename, read_flag));
289	}
290	/* If not running single I/O, just return the access() */
291	return (access(filename, read_flag));
292	}
293	/* If not running single I/O, just return the access() */
294	return (access(filename, read_flag));
295	}
296	/* If not running single I/O, just return the access() */
297	return (access(filename, read_flag));
298	}
299	/* If not running single I/O, just return the access() */
300	return (access(filename, read_flag));
301	}
302	/* If not running single I/O, just return the access() */
303	return (access(filename, read_flag));
304	}
305	/* If not running single I/O, just return the access() */
306	return (access(filename, read_flag));
307	}
308	/* If not running single I/O, just return the access() */
309	return (access(filename, read_flag));
310	}
311	/* If not running single I/O, just return the access() */
312	return (access(filename, read_flag));
313	}
314	/* If not running single I/O, just return the access() */
315	return (access(filename, read_flag));
316	}
317	/* If not running single I/O, just return the access() */
318	return (access(filename, read_flag));
319	}
320	/* If not running single I/O, just return the access() */
321	return (access(filename, read_flag));
322	}
323	/* If not running single I/O, just return the access() */
324	return (access(filename, read_flag));
325	}
326	/* If not running single I/O, just return the access() */
327	return (access(filename, read_flag));
328	}
329	/* If not running single I/O, just return the access() */
330	return (access(filename, read_flag));
331	}
332	/* If not running single I/O, just return the access() */
333	return (access(filename, read_flag));
334	}
335	/* If not running single I/O, just return the access() */
336	return (access(filename, read_flag));
337	}
338	/* If not running single I/O, just return the access() */
339	return (access(filename, read_flag));
340	}
341	/* If not running single I/O, just return the access() */
342	return (access(filename, read_flag));
343	}
344	/* If not running single I/O, just return the access() */
345	return (access(filename, read_flag));
346	}
347	/* If not running single I/O, just return the access() */
348	return (access(filename, read_flag));
349	}
350	/* If not running single I/O, just return the access() */
351	return (access(filename, read_flag));
352	}
353	/* If not running single I/O, just return the access() */
354	return (access(filename, read_flag));
355	}
356	/* If not running single I/O, just return the access() */
357	return (access(filename, read_flag));
358	}
359	/* If not running single I/O, just return the access() */
360	return (access(filename, read_flag));
361	}
362	/* If not running single I/O, just return the access() */
363	return (access(filename, read_flag));
364	}
365	/* If not running single I/O, just return the access() */
366	return (access(filename, read_flag));
367	}
368	/* If not running single I/O, just return the access() */
369	return (access(filename, read_flag));
370	}
371	/* If not running single I/O, just return the access() */
372	return (access(filename, read_flag));
373	}
374	/* If not running single I/O, just return the access() */
375	return (access(filename, read_flag));
376	}
377	/* If not running single I/O, just return the access() */
378	return (access(filename, read_flag));
379	}
380	/* If not running single I/O, just return the access() */
381	return (access(filename, read_flag));
382	}
383	/* If not running single I/O, just return the access() */
384	return (access(filename, read_flag));
385	}
386	/* If not running single I/O, just return the access() */
387	return (access(filename, read_flag));
388	}
389	/* If not running single I/O, just return the access() */
390	return (access(filename, read_flag));
391	}
392	/* If not running single I/O, just return the access() */
393	return (access(filename, read_flag));
394	}
395	/* If not running single I/O, just return the access() */
396	return (access(filename, read_flag));
397	}
398	/* If not running single I/O, just return the access() */
399	return (access(filename, read_flag));
400	}
401	/* If not running single I/O, just return the access() */
402	return (access(filename, read_flag));
403	}
404	/* If not running single I/O, just return the access() */
405	return (access(filename, read_flag));
406	}
407	/* If not running single I/O, just return the access() */
408	return (access(filename, read_flag));
409	}
410	/* If not running single I/O, just return the access() */
411	return (access(filename, read_flag));
412	}
413	/* If not running single I/O, just return the access() */
414	return (access(filename, read_flag));
415	}
416	/* If not running single I/O, just return the access() */
417	return (access(filename, read_flag));
418	}
419	/* If not running single I/O, just return the access() */
420	return (access(filename, read_flag));
421	}
422	/* If not running single I/O, just return the access() */
423	return (access(filename, read_flag));
424	}
425	/* If not running single I/O, just return the access() */
426	return (access(filename, read_flag));
427	}
428	/* If not running single I/O, just return the access() */
429	return (access(filename, read_flag));
430	}
431	/* If not running single I/O, just return the access() */
432	return (access(filename, read_flag));
433	}
434	/* If not running single I/O, just return the access() */
435	return (access(filename, read_flag));
436	}
437	/* If not running single I/O, just return the access() */
438	return (access(filename, read_flag));
439	}
440	/* If not running single I/O, just return the access() */
441	return (access(filename, read_flag));
442	}
443	/* If not running single I/O, just return the access() */
444	return (access(filename, read_flag));
445	}
446	/* If not running single I/O, just return the access() */
447	return (access(filename, read_flag));
448	}
449	/* If not running single I/O, just return the access() */
450	return (access(filename, read_flag));
451	}
452	/* If not running single I/O, just return the access() */
453	return (access(filename, read_flag));
454	}
455	/* If not running single I/O, just return the access() */
456	return (access(filename, read_flag));
457	}
458	/* If not running single I/O, just return the access() */
459	return (access(filename, read_flag));
460	}
461	/* If not running single I/O, just return the access() */
462	return (access(filename, read_flag));
463	}
464	/* If not running single I/O, just return the access() */
465	return (access(filename, read_flag));
466	}
467	/* If not running single I/O, just return the access() */
468	return (access(filename, read_flag));
469	}
470	/* If not running single I/O, just return the access() */
471	return (access(filename, read_flag));
472	}
473	/* If not running single I/O, just return the access() */
474	return (access(filename, read_flag));
475	}
476	/* If not running single I/O, just return the access() */
477	return (access(filename, read_flag));
478	}
479	/* If not running single I/O, just return the access() */
480	return (access(filename, read_flag));
481	}
482	/* If not running single I/O, just return the access() */
483	return (access(filename, read_flag));
484	}
485	/* If not running single I/O, just return the access() */
486	return (access(filename, read_flag));
487	}
488	/* If not running single I/O, just return the access() */
489	return (access(filename, read_flag));
490	}
491	/* If not running single I/O, just return the access() */
492	return (access(filename, read_flag));
493	}
494	/* If not running single I/O, just return the access() */
495	return (access(filename, read_flag));
496	}
497	/* If not running single I/O, just return the access() */
498	return (access(filename, read_flag));
499	}
500	/* If not running single I/O, just return the access() */
501	return (access(filename, read_flag));
502	}
503	/* If not running single I/O, just return the access() */
504	return (access(filename, read_flag));
505	}
506	/* If not running single I/O, just return the access() */
507	return (access(filename, read_flag));
508	}
509	/* If not running single I/O, just return the access() */
510	return (access(filename, read_flag));
511	}
512	/* If not running single I/O, just return the access() */
513	return (access(filename, read_flag));
514	}
515	/* If not running single I/O, just return the access() */
516	return (access(filename, read_flag));
517	}
518	/* If not running single I/O, just return the access() */
519	return (access(filename, read_flag));
520	}
521	/* If not running single I/O, just return the access() */
522	return (access(filename, read_flag));
523	}
524	/* If not running single I/O, just return the access() */
525	return (access(filename, read_flag));
526	}
527	/* If not running single I/O, just return the access() */
528	return (access(filename, read_flag));
529	}
530	/* If not running single I/O, just return the access() */
531	return (access(filename, read_flag));
532	}
533	/* If not running single I/O, just return the access() */
534	return (access(filename, read_flag));
535	}
536	/* If not running single I/O, just return the access() */
537	return (access(filename, read_flag));
538	}
539	/* If not running single I/O, just return the access() */
540	return (access(filename, read_flag));
541	}
542	/* If not running single I/O, just return the access() */
543	return (access(filename, read_flag));
544	}
545	/* If not running single I/O, just return the access() */
546	return (access(filename, read_flag));
547	}
548	/* If not running single I/O, just return the access() */
549	return (access(filename, read_flag));
550	}
551	/* If not running single I/O, just return the access() */
552	return (access(filename, read_flag));
553	}
554	/* If not running single I/O, just return the access() */
555	return (access(filename, read_flag));
556	}
557	/* If not running single I/O, just return the access() */
558	return (access(filename, read_flag));
559	}
560	/* If not running single I/O, just return the access() */
561	return (access(filename, read_flag));
562	}
563	/* If not running single I/O, just return the access() */
564	return (access(filename, read_flag));
565	}
566	/* If not running single I/O, just return the access() */
567	return (access(filename, read_flag));
568	}
569	/* If not running single I/O, just return the access() */
570	return (access(filename, read_flag));
571	}
572	/* If not running single I/O, just return the access() */
573	return (access(filename, read_flag));
574	}
575	/* If not running single I/O, just return the access() */
576	return (access(filename, read_flag));
577	}
578	/* If not running single I/O, just return the access() */
579	return (access(filename, read_flag));
580	}
581	/* If not running single I/O, just return the access() */
582	return (access(filename, read_flag));
583	}
584	/* If not running single I/O, just return the access() */
585	return (access(filename, read_flag));
586	}
587	/* If not running single I/O, just return the access() */
588	return (access(filename, read_flag));
589	}
590	/* If not running single I/O, just return the access() */
591	return (access(filename, read_flag));
592	}
593	/* If not running single I/O, just return the access() */
594	return (access(filename, read_flag));
595	}
596	/* If not running single I/O, just return the access() */
597	return (access(filename, read_flag));
598	}
599	/* If not running single I/O, just return the access() */
600	return (access(filename, read_flag));
601	}
602	/* If not running single I/O, just return the access() */
603	return (access(filename, read_flag));
604	}
605	/* If not running single I/O, just return the access() */
606	return (access(filename, read_flag));
607	}
608	/* If not running single I/O, just return the access() */
609	return (access(filename, read_flag));
610	}
611	/* If not running single I/O, just return the access() */
612	return (access(filename, read_flag));
613	}
614	/* If not running single I/O, just return the access() */
615	return (access(filename, read_flag));
616	}
617	/* If not running single I/O, just return the access() */
618	return (access(filename, read_flag));
619	}
620	/* If not running single I/O, just return the access() */
621	return (access(filename, read_flag));
622	}
623	/* If not running single I/O, just return the access() */
624	return (access(filename, read_flag));
625	}
626	/* If not running single I/O, just return the access() */
627	return (access(filename, read_flag));
628	}
629	/* If not running single I/O, just return the access() */
630	return (access(filename, read_flag));
631	}
632	/* If not running single I/O, just return the access() */
633	

Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM misc/resolve.c	DATE 5/23/89	PAGE # 3/6
LINE #		SOURCE TEXT		
241		p = (char *) getenv (lm_path);		
242		if (NULL == p) {		
243		lm_queue_message(ERROR_MSG, "environment variable \"%s\" is defined ",		
244		lm_path);		
245		return(LM_ERROR);		
246		}		
247		if ((strlen (p) > sizeof (pathname))) {		
248		lm_queue_message(ERROR_MSG, "environment variable \"%s\" is too long ", lm_path);		
249		return(LM_ERROR);		
250		}		
251		(void) strcpy (pathname, p);		
252		p = pathname;		
253		end_of_string = pathname + strlen(pathname);		
254		/* find out if it's a colon or space separated list */		
255		p = (char *) strchr (pathname, ":");		
256		if (NULL == p) p = pathname + strlen(pathname);		
257		(void) strcpy (delimiter, p, 1);		
258		delimiter[1] = NULL;		
259		/* look for the file in each element of the path */		
260		do {		
261		p = (char *) strchr (p, delimiter);		
262		if (NULL == p) p = pathname + strlen(pathname);		
263		(void) strcpy (p, p, p);		
264		dir(p) = NULL;		
265		p = p + 1;		
266		(void) strcpy (dir, "/");		
267		(void) strcpy (dir, filename);		
268		accessible = lm_access(dir, (int) read_flag);		
269		if ((accessible == 0) && (!file_is_a_directory(dir))) {		
270		(void) strcpy (return_filename, dir, (int) filename_size);		
271		return (LM_SUCCESS);		
272		} while (p != end_of_string);		
273		/* if we get here, the file does not exist */		
274		/* which is OK if we're going to create it */		
275		if (2 == read_flag) {		
276		/* return the first directory that we can write to */		
277		p = (char *) getenv (lm_path);		
278		(void) strcpy (pathname, p);		
279		p = pathname;		
280		p = (char *) strchr (pathname, ":");		
281		if (NULL == p) p = pathname + strlen(pathname);		
282		(void) strcpy (delimiter, p, 1);		
283		delimiter[1] = NULL;		
284		do {		
285		p = (char *) strchr (p, delimiter);		
286		if (NULL == p) p = pathname + strlen(pathname);		
287		(void) strcpy (dir, p, p);		
288		dir(p) = NULL;		
289		p = p + 1;		
290		accessible = lm_access(dir, (int) read_flag);		
291		if (accessible == 0) {		
292		(void) strcpy (dir, "/");		
293		(void) strcpy (dir, filename);		
294		(void) strcpy (return_filename, dir, (int) filename_size);		
295		return (LM_SUCCESS);		
296		} while (p != end_of_string);		
297		/* if we get here, we couldn't find a directory to write to */		
298		lm_queue_message (ERROR_MSG,		
299		"write permission denied on file \"%s\" in \"%s\" ",		
300		filename, lm_path);		
301		return(LM_ERROR);		
302		}		
303		lm_queue_message(ERROR_MSG,		
304		"unable to find file \"%s\" in \"%s\" ", filename, lm_path);		
305		return (LM_ERROR);		
306		}		
307		lm_resolve_library_file(lm_path, target_filename, return_filename, filename_size)		
308		char *lm_path, *target_filename, *return_filename,		
309		long filename_size;		
310		{		
311		char *p, *pos, dir[max_str], pathname[1024];		
312		char file_name[max_str];		
313		char end_of_string;		
314		long accessible, max_files;		
315		char delimiter[max_str];		
316		/* first, check to see if the file is in the default directory */		
317		accessible = lm_access(target_filename, 0);		
318		(void) strcpy (return_filename, target_filename, (int) filename_size);		
319		if ((accessible == 0) && (!file_is_a_directory(target_filename))) return(LM_SUCCESS);		
320		/* Now, let's look down the LM_DIR path */		
321		p = (char *) getenv (lm_path);		
322		if (NULL == p) {		
323		lm_queue_message(ERROR_MSG, "environment variable \"%s\" is not defined ",		
324		lm_path);		
325		return(LM_ERROR);		



Copyright 1989 Logic Modeling Systems		SOURCE PROGRAM	#	DATE	5/23/89	PAGE #
		misc/resolve.c		TIME	1:20:42 pm	4/7
LINE #	SOURCE TEXT					
361	}					
362	if ((strlen(p) > sizeof(pathname))) {					
363	ln_queue_message(ERROR_MSG, "environment variable \"%s\" is too long", ln_path);					
364	return(LM_ERROR);					
365	}					
366	(void) strcpy(pathname, p);					
367	p = pathname;					
368	pos = (char *) strchr(pathname, "/");					
369	if (NULL == pos) pos = pathname + strlen(pathname);					
370	(void) strcpy(delimiter, pos, 1);					
371	delimiter[1] = NULL;					
372	/* look in each of the elements of the path */					
373	end_of_string = pathname + strlen(pathname);					
374	do {					
375	pos = (char *) strchr(p, delimiter);					
376	if (NULL == pos) pos = pathname + strlen(pathname);					
377	(void) strcpy(dir, p, pos-p);					
378	dir[pos-p] = NULL;					
379	p = pos + 1;					
380	(void) strcpy(file_name, dir);					
381	(void) strcat(file_name, "/");					
382	(void) strcat(file_name, target_filename);					
383	accessible = ln_access(file_name, 0);					
384	if ((accessible == 0) && (!file_is_a_directory(file_name))) {					
385	(void) strcpy(return_filename, file_name, (int) filename_size);					
386	/* we have found the file we're looking for in this */					
387	/* directory. Now, check for the same file in a subdire */					
388	/* if we find it, it's an error */					
389	if (0 != check_all_sub_dirs(dir, target_filename, return_filename, filename_size)) {					
390	ln_queue_message(ERROR_MSG, "%s exists in more than one subdirectory of %s",					
391	target_filename, dir);					
392	return(LM_ERROR);					
393	}					
394	return(LM_SUCCESS);					
395	}					
396	/* We couldn't find the file in the directory, now check all */					
397	/* subdirectories */					
398	if (1 == (num_files = check_all_sub_dirs(dir, target_filename, return_filename, filename_size)))					
399	return(LM_SUCCESS);					
400	if (num_files > 1) {					
401	if (0 != check_all_sub_dirs(dir, target_filename, return_filename, filename_size)) {					
402	ln_queue_message(ERROR_MSG, "%s exists in more than one subdirectory of %s",					
403	target_filename, dir);					
404	return(LM_ERROR);					
405	}					
406	}					
407	} while (pos != end_of_string);					
408	ln_queue_message(ERROR_MSG, "unable to find file \"%s\" in \"%s\"", target_filename, ln_path);					
409	return(LM_ERROR);					
410	}					
411	static long					
412	check_all_sub_dirs(dir, filename, return_file, ret_size)					
413	char					
414	"dir",					
415	"filename",					
416	"return_file",					
417	"ret_size",					
418	long					
419	num_files = 0;					
420	char					
421	file[max_strl,					
422	DIR					
423	dir_pointer,					
424	char					
425	found_dir_name,					
426	#ifdef SYS5					
427	struct direct *dir_entry,					
428	#else					
429	struct direct *dir_entry,					
430	#endif					
431	dir_pointer = opendir(dir);					
432	if (NULL == dir_pointer) {					
433	ln_queue_message(WARNING_MSG, "Unable to open directory for reading: %s", dir);					
434	return(-1);					
435	}					
436	while (NULL != (dir_entry = readdir(dir_pointer))) {					
437	#ifdef SYS7					
438	found_dir_name = dir_entry->d_name + kldge_spollo_sys5_ar9_7();					
439	#else					
440	/* all SYS7 systems */					
441	found_dir_name = dir_entry->d_name;					
442	#endif					
443	/* SYS7 */					
444	if ((strcmp(found_dir_name, ".") != 0) &&					
445	(strcmp(found_dir_name, "..") != 0)) {					
446	(void) strcpy(file, dir);					
447	(void) strcat(file, "/");					
448	(void) strcat(file, found_dir_name);					
449	(void) strcat(file, "/");					
450	(void) strcat(file, filename);					
451	if (0 == ln_access(file, 0) && (!file_is_a_directory(file))) {					
452	num_files++;					
453	if (num_files == 1) {					
454	(void) strcpy(return_file, file, (int) ret_size);					
455	if (ret_size < strlen(file)) {					
456	ln_queue_message(ERROR_MSG,					
457	"Expanded filename too long for buffer");					
458	return(-1);					
459	}					
460	}					

Copyright 1989 Logic Modeling Systems	SOURCE PROGRAM	DATE	PAGE #
	misc/resolve.c	5/23/89	5/8
		TIME	1:20:42 pm

LINE #	SOURCE TEXT
481	}
482	
483	(void) closedir (dir_pointer);
484	
485	return (new_files);
486	
487	}
488	
489	#ifdef SH9_7
490	/*
491	* This routine determines whether we are executing in a STS or BSD environment
492	* by exploiting the fact that on STS sprintf() returns a reasonable value
493	* that equals the number of characters transferred into the buffer. This is
494	* opposed to BSD which returns the address of the output buffer. We check to
495	* see if sprintf() doesn't return the address of the buffer, thus indicating
496	* a STS runtime environment.
497	* This information is used as components for the difference in the definition
498	* of the structure returned by readdir() in the STS and BSD environments.
499	* A lucky(?) turn of events is that in SH9-1, readdir() returns a structure
500	* that is equivalent in both environments.
501	*/
502	static
503	kludge_spello_sys_sh9_7()
504	{
505	char buf[21];
506	
507	if ((int)sprintf(buf, "A") != (int)buf)
508	return(2); /* STS */
509	else return(0); /* BSD */
510	}
511	#endif /* SH9_7 */
512	
513	static file_is_a_directory(filename)
514	char *filename;
515	
516	{
517	struct stat filestat_buffer;
518	
519	stat(filename, &filestat_buffer);
520	if (filestat_buffer.st_mode & S_IFDIR) return (SUCCESS);
521	
522	return (FAILURE);
523	}
524	
525	/*else /* VMS */
526	
527	#include "common.h"
528	#include "message.h"
529	#include "lm_sfi.h"
530	#include "acdef.h"
531	#include "rnode.h"
532	#include "descrip.h"
533	
534	#define max_str 256
535	
536	lm_check_path_variable (lm_path)
537	char *lm_path;
538	
539	{
540	
541	if (NULL == (char *) getenv (lm_path)) {
542	lm_message(ERROR_MSG, "environment variable \"%s\" is not defined ",
543	lm_path);
544	return(LM_ERROR);
545	}
546	
547	return(LM_SUCCESS);
548	}
549	
550	lm_resolve_filename(lm_path, filename, return_filename, filename_size, read_flag)
551	char *lm_path, *filename, *return_filename;
552	long filename_size, read_flag;
553	
554	{
555	int context;
556	int status;
557	struct dscdescriptor filespec_descriptor;
558	struct dscdescriptor result_descriptor;
559	char filespec(max_str);
560	struct {
561	short size;
562	char result(max_str);
563	ret_file;
564	}
565	
566	status = lm_check_path_variable (lm_path);
567	if (status != LM_SUCCESS)
568	return(status);
569	
570	strcpy (filespec, filename);
571	
572	filespec_descriptor.dscb_class = DSCB_CLASS_2;
573	filespec_descriptor.dscb_dtype = DSCB_DTYPE_T;
574	filespec_descriptor.dscb_length = strlen(filespec);
575	filespec_descriptor.dscb_pointer = filespec;
576	
577	result_descriptor.dscb_class = DSCB_CLASS_V2;
578	result_descriptor.dscb_dtype = DSCB_DTYPE_T;
579	result_descriptor.dscb_length = sizeof(ret_file.result);
580	result_descriptor.dscb_pointer = &ret_file;
581	context = 0;
582	
583	status = libsfnd_file (filespec_descriptor, &result_descriptor, &context);
584	
585	if ((DSS_NORMAL == status)    (DSS_NORMAL == status)) {
586	ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */
587	strcpy (return_filename, ret_file.result, filename_size);
588	return(LM_SUCCESS);
589	}
590	
591	strcpy (filespec, lm_path);
592	strcat (filespec, "/");
593	strcat (filespec, filename);
594	
595	if (read_flag != 0) {
596	strcpy (return_filename, filespec, filename_size);
597	return(LM_SUCCESS);
598	}
599	

COPYRIGHT 1989 Logic Modeling Systems		SOURCE PROGRAM misc/resolve.c	DATE 5/23/89	PAGE # 6/9
LINE #	SOURCE TEXT			
601	<pre> filespec_descriptor.dcsb_class = DCSK_CLASS_S, filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T, filespec_descriptor.dcsb_length = strlen(filespec), filespec_descriptor.dcsb_pointer = filespec, result_descriptor.dcsb_class = DCSK_CLASS_VS, result_descriptor.dcsb_dtyp = DCSK_DTYPE_T, result_descriptor.dcsb_length = sizeof(ret_file.result), result_descriptor.dcsb_pointer = &amp;ret_file, count = 0,  status = libfind_file (filespec_descriptor, &amp;result_descriptor, &amp;context), if (((RMS_NORMAL == status)    (RMS_NORMAL == status))) {     in_queue_message (VMS_ERROR_MSG, status,         "unable to find file \"%s\" in \"%s\" ", filename,         in_path);     return (LM_ERROR); }  ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */ strcpy (return_filename, ret_file.result, filename_size); return (LM_SUCCESS); }  lib_resolve_library_file (in_path, target_filename, return_filename, filename_size) char *in_path, *target_filename, *return_filename, long filename_size; {     int count = 0;     int status;     int count;     char *p;     int i;      struct dcsdescriptor filespec_descriptor;     struct dcsdescriptor result_descriptor;     char filespec[max_str];     struct {         short size;         char result[max_str];         ret_file;     }     strcpy (filespec, target_filename);     filespec_descriptor.dcsb_class = DCSK_CLASS_S,     filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T,     filespec_descriptor.dcsb_length = strlen(filespec),     filespec_descriptor.dcsb_pointer = filespec,     result_descriptor.dcsb_class = DCSK_CLASS_VS,     result_descriptor.dcsb_dtyp = DCSK_DTYPE_T,     result_descriptor.dcsb_length = sizeof(ret_file),     result_descriptor.dcsb_pointer = &amp;ret_file,     status = libfind_file (filespec_descriptor, &amp;result_descriptor, &amp;context),     if (((RMS_NORMAL == status)    (RMS_NORMAL == status))) {         ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */         strcpy (return_filename, ret_file.result, filename_size);         return (LM_SUCCESS);     }     else {         strcpy (filespec, in_path);         struct {             char *p;             char *target_filename;         }         filespec_descriptor.dcsb_class = DCSK_CLASS_S,         filespec_descriptor.dcsb_dtyp = DCSK_DTYPE_T,         filespec_descriptor.dcsb_length = strlen(filespec),         filespec_descriptor.dcsb_pointer = filespec,         count = 0;         count = 0;         do {             status = libfind_file (filespec_descriptor, &amp;result_descriptor, &amp;context);             if (((RMS_NORMAL == status)    (RMS_NORMAL == status))) {                 ret_file.result[ret_file.size] = '\0'; /* NULL terminate the string */                 strcpy (return_filename, ret_file.result, filename_size);                 count++;             }             while (count != 0);             if (count == 0) {                 in_queue_message (VMS_ERROR_MSG, status,                     "unable to find file \"%s\" in \"%s\" ",                     target_filename, in_path);                 return (LM_ERROR);             }             if (count &gt; 1) {                 in_queue_message (ERROR_MSG,                     "%s exists in more than one subdirectory of %s ",                     target_filename, in_path);                 return (LM_ERROR);             }             if (count == 1) {                 for (i=0; p=return_filename; i=strlen(return_filename), i++, p++)                     if (0 == strcmp(p, "[000000.", 9))                         strcpy (p, p+9);                 for (i=0; p=return_filename; i=strlen(return_filename), i++, p++)                     if (0 == strcmp(p, "[000000.", 8))                         strcpy (p, p+8);                 return (LM_SUCCESS);             }         }     } } #endif </pre>			

We claim:

1. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the

65

hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin according to one of a pre-determined soft-drive high I/V characteristic curve and



programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first reference voltage and a second reference voltage to indicate that the pin is in the non-driving state.

8. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a first reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is between the first reference voltage and a second reference voltage to indicate that the pin is in the non-driving state.

9. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than the reference current to indicate that the pin is in a driving low state, or if the current into the pin is less than the reference current to indicate that the pin is in another state.

10. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

ware means to drive the pin according to one of a predetermined soft-drive high I/V characteristic curve and a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

11. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

12. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a reference current to indicate that the pin is in a driving high state, or if the current into the pin is greater than the reference current to indicate that the pin is in another state.

13. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state,

non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

14. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

15. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive low I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive low I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driv-

ing low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

16. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin according to a predetermined soft-drive high I/V characteristic curve or a predetermined soft-drive low I/V characteristic curve, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin according to the predetermined soft-drive high I/V characteristic curve;
- (b) driving the pin with the pin driver of the hardware modeling system according to the predetermined soft-drive high I/V characteristic curve; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a first reference current to indicate that the pin is in the driving high state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

17. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein the method further comprises the step of selecting one of the soft-drive low I/V characteristic curve and the soft-drive high I/V characteristic curve to program the pin driver according to a state of simulated circuitry connected to the pin in a simulated circuit design.

18. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the method further comprises the step of selecting one of the soft-drive low I/V characteristic curve and the soft-drive high I/V characteristic curve to program the pin driver according to a state of simulated circuitry connected to the pin in a simulated circuit design.

19. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, or 16, wherein the method further comprises the step of determining simultaneously the states of a plurality of pins of an electronic device or circuitry that are electrically coupled to software programmable pin drivers of the hardware modeling systems.

20. The method as recited in claim 19, wherein the pin drivers that are electrically coupled to the pins are collectively programmable to drive with the predetermined soft-drive high I/V characteristic curve or predetermined soft-drive low I/V characteristic curve.

21. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein the reference voltages are software programmable by the hardware modeling system.

22. The method as recited in claim 17, wherein the reference voltages are software programmable by the hardware modeling system.

23. The method as recited in claim 19, wherein the reference voltages are software programmable by the hardware modeling system.

24. The method as recited in claim 20, wherein the reference voltages are software programmable by the hardware modeling system.

25. The method as recited in claim 1, 2, 3, 4, 5, 6, 7, or 8, wherein reference voltages are hardware programmable by the hardware modeling system.



26. The method as recited in claim 17, wherein the reference voltages are hardware programmable by the hardware modeling system.

27. The method as recited in claim 19, wherein the reference voltages are hardware programmable by the hardware modeling system.

28. The method as recited in claim 20, wherein the reference voltages are hardware programmable by the hardware modeling system.

29. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the reference currents are software programmable by the hardware modeling system.

30. The method as recited in claim 18, wherein the reference currents are software programmable by the hardware modeling system.

31. The method as recited in claim 19, wherein the reference currents are software programmable by the hardware modeling system.

32. The method as recited in claim 20, wherein the reference currents are software programmable by the hardware modeling system.

33. The method as recited in claim 9, 10, 11, 12, 13, 14, 15, or 16, wherein the reference currents are hardware programmable by the hardware modeling system.

34. The method as recited in claim 18, wherein the reference currents are hardware programmable by the hardware modeling system.

35. The method as recited in claim 19, wherein the reference currents are hardware programmable by the hardware modeling system.

36. The method as recited in claim 20, wherein the reference currents are hardware programmable by the hardware modeling system.

37. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is above the reference voltage to indicate that the pin is in another state.

38. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is between a first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is other than between the first and second reference voltages to indicate that the pin is in another state.

39. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward one of a logic low and a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is between a first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is other than between the first and second reference voltages to indicate that the pin is in another state.

40. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin towards a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is below the reference voltage to indicate that the pin is in another state.

41. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is above the second reference voltage to indicate that the pin is in the driving high state.

42. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between a first

and a second reference voltage to indicate that the pin is in the non-driving state, or if the voltage of the pin is above the second reference voltage to indicate that the pin is in the driving high state.

43. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the voltage of the pin is below a first reference voltage to indicate that the pin is in the driving low state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state.

44. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the voltage of the pin is above a first reference voltage to indicate that the pin is in the driving high state, or if the voltage of the pin is between the first and a second reference voltage to indicate that the pin is in the non-driving state.

45. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in the driving low state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a reference current to indicate that the pin is in the driving low state, or if the current into the pin is less than the reference current to indicate that the pin is in another state.

46. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first reference

current and a second reference current to indicate that the pin is in the non-driving state, or if the current at the pin is other than between the first and second reference currents to indicate that the pin is in another state.

47. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is between a first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is other than between the first and second reference currents to indicate that the pin is in another state.

48. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic low or a logic high, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a reference current to indicate that the pin is in the driving high state, or if the current into the pin is greater than the reference current to indicate that the pin is in another state.

49. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

50. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state, non-driving state, or driving high state, the pin driver being individually programmable by software means to



drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state, or if the current into the pin is less than the second reference current to indicate that the pin is in the driving high state.

51. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving low state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic low;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic low; and
- (c) while driving the pin, automatically determining if the current into the pin is greater than a first reference current to indicate that the pin is in the driving low state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

52. A method for use in a hardware modeling system for determining if a pin of an electronic device or circuitry that is electrically coupled to a pin driver of the hardware modeling system is in a driving high state or non-driving state, the pin driver being individually programmable by software means to drive the pin toward a logic high or a logic low, comprising the steps of:

- (a) programming the pin driver of the hardware modeling system to drive the pin toward the logic high;
- (b) driving the pin with the pin driver of the hardware modeling system toward the logic high; and
- (c) while driving the pin, automatically determining if the current into the pin is less than a first reference current to indicate that the pin is in the driving high state, or if the current into the pin is between the first and a second reference current to indicate that the pin is in the non-driving state.

53. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the voltage of the pin with a plurality of reference voltage levels that differentiate between at least four states of the pin.

54. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with a plurality of reference current levels that differentiate between at least four states of the pin.

55. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware

modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time; automatically comparing the voltage at the pin with a plurality of reference voltage levels that differentiate between at least a driving low state, non-driving low state, non-driving high state, and driving high state.

56. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with a plurality of reference current levels that differentiate between at least a driving low state, non-driving low state, non-driving high state, and driving high state.

57. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the voltage at the pin with at least three reference voltages.

58. The method as recited in claim 57, wherein the method further includes the step of programming the reference voltages with the hardware modeling system.

59. The method as recited in claim 58, wherein the method includes programming the reference voltages by software means.

60. The method as recited in claim 58, wherein the method includes programming the reference voltages by hardware means.

61. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin with at least three reference currents.

62. The method as recited in claim 61, wherein the method further includes the step of programming the reference currents with the hardware modeling systems.

63. The method as recited in claim 62, wherein the method includes programming the reference currents by software means.

64. The method as recited in claim 62, wherein the method includes programming the reference currents by hardware means.

65. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing voltage at the pin in at least three comparators with a different reference voltage being provided to each comparator.

66. The method as recited in claim 65, wherein the reference voltages are software programmable by the hardware modeling system.

67. The method as recited in claim 65, wherein the reference voltages are hardware programmable by the hardware modeling system.

68. A method for determining with a hardware modeling system a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system and at the same time, automatically comparing the current into the pin

in at least three comparators with a different reference current being provided to each comparator.

69. The method as recited in claim 68, wherein the reference currents are software programmable by the hardware modeling system.

70. The method as recited in claim 68, wherein the reference currents are hardware programmable by the hardware modeling system.

71. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, with the pin driver being programmable to drive with any one of at least eight different current limits, and at the same time, automatically comparing the voltage of the pin with at least one reference voltage.

72. The method as recited in claim 71 wherein the electronic devices or circuitry includes a plurality of pins electrically coupled to pin drivers of the hardware modeling system, and the current limit of the pin driver associated with each pin is independently software programmable by the hardware modeling system.

73. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, with the pin driver being programmable to drive with any one of at least eight different current limits and at the same time, automatically comparing the current into the pin with at least one reference current.

74. The method as recited in claim 73, wherein the electronic devices or circuitry includes a plurality of pins electrically coupled to pin drivers of the hardware modeling system, and the current limit of the pin driver associated with each pin is independently software programmable by the hardware modeling system.

75. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, the pin driver being both a current sink and source and driving toward a predetermined software programmable voltage level, and at the same time, automatically comparing the voltage of the pin with at least two reference voltages.

76. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable, current limited pin driver of the hardware modeling system, the pin driver being both a current sink and source and driving toward a predetermined software programmable voltage level and at the same time, automatically comparing the current into the pin with at least two reference currents.

77. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable pin driver of the hardware modeling system, the pin driver being programmable to drive with any one of at least eight different I/V characteristic curves, and at the same time, automatically

comparing the voltage of the pin with at least one reference voltage.

78. A method for use in a hardware modeling system for determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a software programmable pin driver of the hardware modeling system, the pin driver being programmable to drive with any one of at least eight different I/V characteristic curves, and at the same time, automatically comparing the current into the pin with at least one reference current.

79. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system in response to an access to the hardware modeling system being assessed by a simulator via network means.

80. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining a state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system with the hardware modeling system being time shared among a plurality of simulators.

81. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the state of a pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, the hardware modeling system having a plurality of electronic devices or circuitry connected thereto with each electronic device or circuitry having at least one pin that is electrically coupled to the hardware modeling system.

82. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the pin is capable of functioning as both an input and an output.

83. The method as recited in claims 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, wherein the method includes determining the states of a plurality of pins of the electronic device or circuitry simultaneously.

84. The method as recited in claim 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, or 78, wherein the method further includes as a first step resetting and restoring the electronic device or circuitry to a specific internal state by presentation of a history sequence of stimulation patterns to the electronic device or circuitry.

85. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one current limited pin driver of the hardware modeling system that is electrically coupled to a pin of the electronic device or circuitry, the current limit of the pin driver of the hardware modeling system being software programmable by the hardware modeling system.

86. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one pin driver of the hardware modeling system that is electrically coupled to a pin of the electronic device or circuitry, the pin driver of the hardware mod-

eling system providing a programmable drive voltage for driving the pin connected thereto, the drive voltage being programmable to at least three different voltages.

87. The multi-channel pin driver integrated circuit as recited in claim 86, wherein the drive voltage is software programmable by the hardware modeling system.

88. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one channel having at least three voltage comparators for use in determining a state of at least one pin.

89. A multi-channel pin driver integrated circuit of a hardware modeling system for simultaneously stimulating and sensing at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including at least one channel having at least one voltage comparator for use in determining the state of at least one pin, the voltage comparator having input thereto a programmable reference voltage.

90. The multi-channel pin driver integrated circuit as recited in claim 89, wherein the reference voltage is software programmable by the hardware modeling system.

91. A method for use in a hardware modeling system for executing hardware modeling system diagnostics using known-good diagnostic circuitry, comprising the steps of:

- (a) electrically coupling the diagnostic circuitry to the pin electronics circuitry of the hardware modeling system;
- (b) presenting stimulus patterns to the diagnostic circuitry using the pin electronics circuitry;
- (c) measuring a response of the diagnostic circuitry to the stimulus patterns applied in step (b); and
- (d) automatically comparing the response measured in step (c) with a known-good response to determine if the hardware modeling system is functioning properly.

92. A method for use in a hardware modeling system for determining a state of at least one pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system, and at the same time, automatically determining whether the pin is in a driving low state or a non-driving low state.

93. A method for use in a hardware modeling system for determining a state of at least one pin of an electronic device or circuitry that is electrically coupled to the hardware modeling system, comprising: driving the pin with a pin driver of the hardware modeling system, and at the same time, automatically determining whether the pin is in a driving high state or a non-driving high state.

94. A method for use in a hardware modeling system for restoring an electronic device or circuitry to a specific internal state, the electronic device or circuitry having at least one I/O pin electrically coupled to the hardware modeling system, the method comprising presenting a history sequence of stimulation patterns to the I/O pin of the electronic device or circuitry with a current limited pin driver of the hardware modeling system coupled to the I/O pin.

95. The method as recited in claim 94, wherein the current limit for the pin driver is less than 50 mA.

96. A method for use in a hardware modeling system for restoring an electronic device or circuitry to a specific internal state, the electronic device or circuitry having at least one I/O pin electrically coupled to the hardware modeling system, the method comprising presenting a history sequence of stimulation patterns to the I/O pin of the electronic device or circuitry with a pulsed driver of the hardware modeling system coupled to the I/O pin.

97. A method for use in a hardware modeling system for determining an output delay of at least a first pin of an electronic device or circuitry according to a present internal state of the electronic device or circuitry in response to a stimulus applied to at least a second pin of the electronic device or circuitry, the pins being electrically coupled to the hardware modeling system, comprising the steps of:

- (a) resetting the electronic device or circuitry coupled to the hardware modeling system to a known internal state;
- (b) restoring the electronic device or circuitry coupled to the hardware modeling system to the present internal state;
- (c) stimulating the electronic device or circuitry coupled to the hardware modeling system by applying stimulus through at least the second pin;
- (d) sampling the state of the first pin according to a software programmable delay after the stimulus was applied;
- (e) changing the software programmable delay in a predetermined manner based on the state sampled during step (d); and
- (f) repeating steps (a)–(e) until the output delay time is determined.

98. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including circuit means to stage pin stimulation patterns for simultaneous presentation of the pin stimulation patterns to the pins connected to the integrated circuit and for simultaneous presentation of the pin stimulation patterns with pin stimulation patterns of other multi-channel pin driver integrated circuits.

99. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including one shared data path for both programming internal control registers of the integrated circuit and for providing pin stimulation pattern data to the integrated circuit from circuitry of the hardware modeling system.

100. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including one shared data path for both reading internal control registers of the integrated circuit and for providing pin stimulation pattern data to the integrated circuit from circuitry of the hardware modeling system.

101. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including circuitry for error checking incoming pins stimulation patterns.

102. The multi-channel pin driver integrated circuit as recited in claim 101, wherein the error checking circuitry performs parity error checks.

103. A multi-channel pin driver integrated circuit of a hardware modeling system for stimulating at least a portion of the pins of an electronic device or circuitry electrically coupled to the integrated circuit, the integrated circuit including a strobe input for use in determining the sampling times for measuring output delays.

104. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having at least one I/O pin, the I/O pin being electrically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) storing a history sequence of stimulation patterns for the I/O pin in memory means with at least one stimulation pattern of the history sequence for the I/O pin being stored in a single bit of the memory means;
- (b) retrieving from the memory means at least one stimulation pattern of the history sequence for the I/O pin;
- (c) presenting the retrieved I/O pin stimulation pattern or patterns to the I/O pin with the pin driver of the hardware modeling system; and
- (d) repeating steps (b) and (c) until the entire history sequence is presented to the I/O pin.

105. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having at least one I/O pin, the I/O pin being electronically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) storing a history sequence of stimulation patterns for the I/O pin in memory means in a plurality of memory means bits, the number of bits being used being less than twice the number of stimulation patterns for the I/O pin;
- (b) retrieving from the memory means at least one stimulation pattern of the history sequence for the I/O pin;
- (c) presenting the retrieved I/O pin stimulation pattern or patterns to the I/O pin with the pin driver of the hardware modeling system; and
- (d) repeating steps (b) and (c) until the entire history sequence is presented to the pin.

106. A method for use in a hardware modeling system of restoring an internal state of an electronic device or circuitry having pins electrically coupled to pin drivers of the hardware modeling system and at least one I/O pin that is electrically coupled to a pin driver of the hardware modeling system, comprising the steps of:

- (a) presenting a history sequence of stimulation patterns to the pins of the electronic device or circuitry with the pin drivers of the hardware modeling system;
- (b) determining the state of the I/O pin;
- (c) embedding in the history sequence a stimulation pattern which will cause the pin driver of the hardware modeling system connected to the I/O pin to

drive high when said pattern of the history sequence is again presented regardless of the logic state of simulated circuitry connected to the I/O pin in a design under simulation if the state of the I/O pin was determined in step (b) to be a driving high state; and

- (d) embedding in the history sequence a pin stimulation pattern which will cause the pin driver of the hardware modeling system connected to the I/O pin to drive in a low state when said pattern of the history sequence is again presented regardless of the logic state of a simulated circuitry connected to the I/O pin in a design under simulation if the state of the I/O pin was determined in step (b) to be a driving low state.

107. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the electronic device or circuitry being electrically coupled to the hardware modeling system, the improvement being that the presence and type of electronic device or circuitry is determined automatically when the electronic device or circuitry is connected to a powered hardware modeling system.

108. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the improvement being that the hardware modeling system has fixturing means for connecting an electronic device or circuitry to the hardware modeling system that is powered.

109. An apparatus for connecting an electronic device or circuitry to a hardware modeling system, comprising fixturing means that has matched length traces connecting pins of the electronic device or circuitry to pin electronics circuitry of the hardware modeling system.

110. A method for use in a hardware modeling system for generating a portion of an electronic device or circuitry timing specification, the electronic device or circuitry being electrically coupled to the hardware modeling system, the method comprising measuring output delays of pins of the electrical device or circuitry coupled to the hardware modeling system, in response to stimulus, and deriving the timing specification from the output delays.

111. A hardware modeling system for stimulating and sensing a response of electronic devices or circuitry to the stimulus, the hardware modeling system having a plurality of electronic devices or circuitry electrically coupled thereto, the improvement being that a fixturing means of the hardware modeling system provides a plurality of different power supply voltages to accommodate connections of electronic devices or circuitry that operate at different power supply voltages.

\* \* \* \* \*

UNITED STATES PATENT AND TRADEMARK OFFICE  
**CERTIFICATE OF CORRECTION**

PATENT NO. : 5,353,243  
DATED : October 4, 1994  
INVENTOR(S) : Andrew J. Read et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 1863, line 48, change "low" to --high--.  
Column 1863, line 58, insert --the-- after "that".  
Column 1865, line 59, change "that" to --than--.  
Column 1865, line 62, insert --that-- after "indicate".  
Column 1868, line 67, insert --the-- after "wherein".  
Column 1871, line 52, change "circuit" to --current--.  
Column 1874, lines 42-43, change "systems" to  
--system--.  
Column 1875, line 58, insert a comma --,-- immediately  
after "level".  
Column 1878, line 68, change "pins" to --pin--.

Signed and Sealed this  
Twenty-first Day of March, 1995

Attest:



BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks